

# CSE 250

## Data Structures

Dr. Eric Mikida  
epmikida@buffalo.edu  
208 Capen Hall

# Lec 26: Balancing Trees

# Announcements

- PA2 due Wednesday
- Midterm #2 next Friday, more info coming soon

# Collection ADTs

Property	Seq	Set	Bag
Explicit Order	✓		
Enforced Uniqueness		✓	
Iterable	✓	✓	✓

# BST Operations

Operation	Runtime
find	$O(d)$
insert	$O(d)$
remove	$O(d)$

# BST Operations

Operation	Runtime
find	$O(d)$
insert	$O(d)$
remove	$O(d)$

*What is the runtime in terms of  $n$ ?  $O(n)$*

*What about the lower bound?  $\Omega(\log(n))$*

*Can we do better? **TBD...***

# Collection ADTs

Property	Seq	Set	Bag
Explicit Order	✓		
Enforced Uniqueness		✓	
<b>Iterable</b>	✓	✓	✓

# Tree Traversals

**Goal:** Visit every element of a tree (in linear time?)

## **Pre-Order (top-down)**

Visit the **root**, then the **left** subtree, then the **right** subtree

## **In-Order**

Visit the **left** subtree, then the **root**, then the **right** subtree

## **Post-Order (bottom-up)**

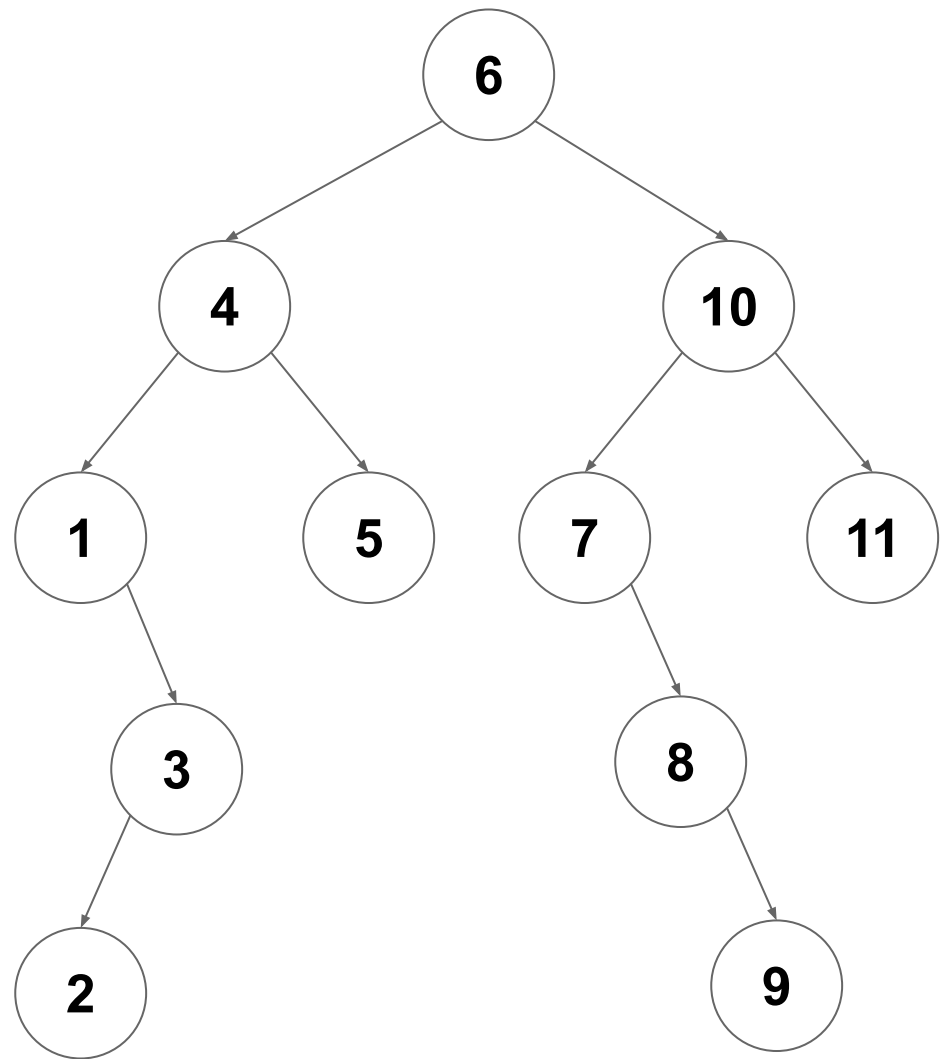
Visit the **left** subtree, then the **right** subtree, then the **root**

# Tree Traversal: In-Order

```
1 void inOrderVisit(Optional<TreeNode<T>> root, Visitor v) {  
2     if (root.isPresent()) {  
3         inOrderVisit(root.get().leftChild, v);  
4         v.visit(root.get().value);  
5         inOrderVisit(root.get().rightChild, v);  
6     }  
7 }
```

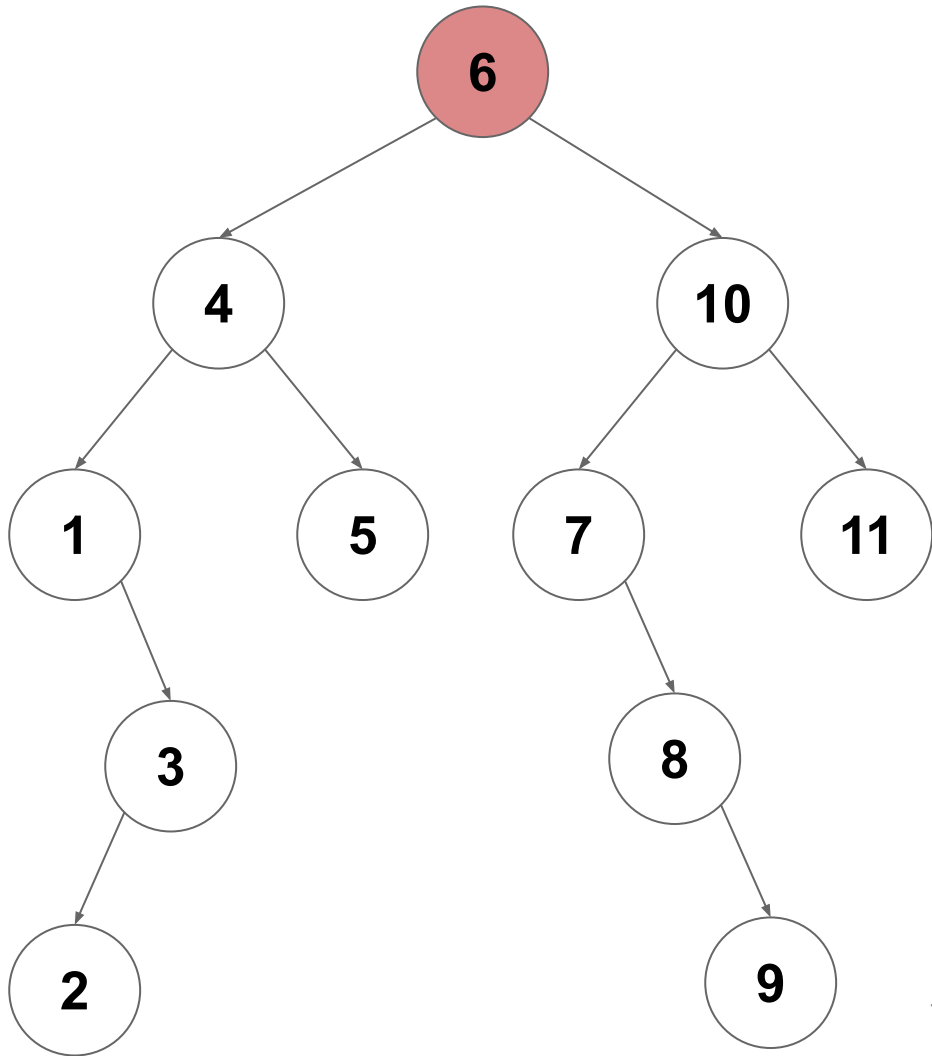


# In-Order Traversal on a BST



# In-Order Traversal on a BST

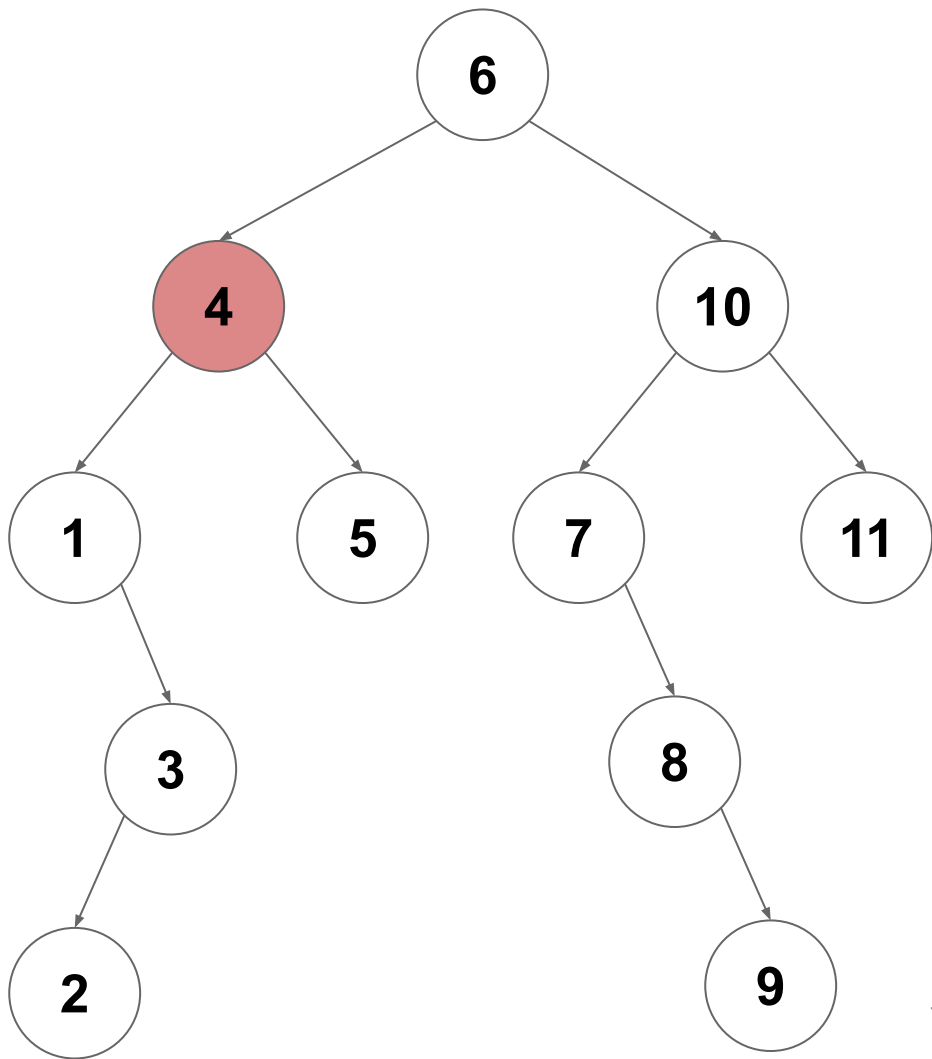
`inorderVisit(6)`



# In-Order Traversal on a BST

`inorderVisit(6)`

`inorderVisit(4)`

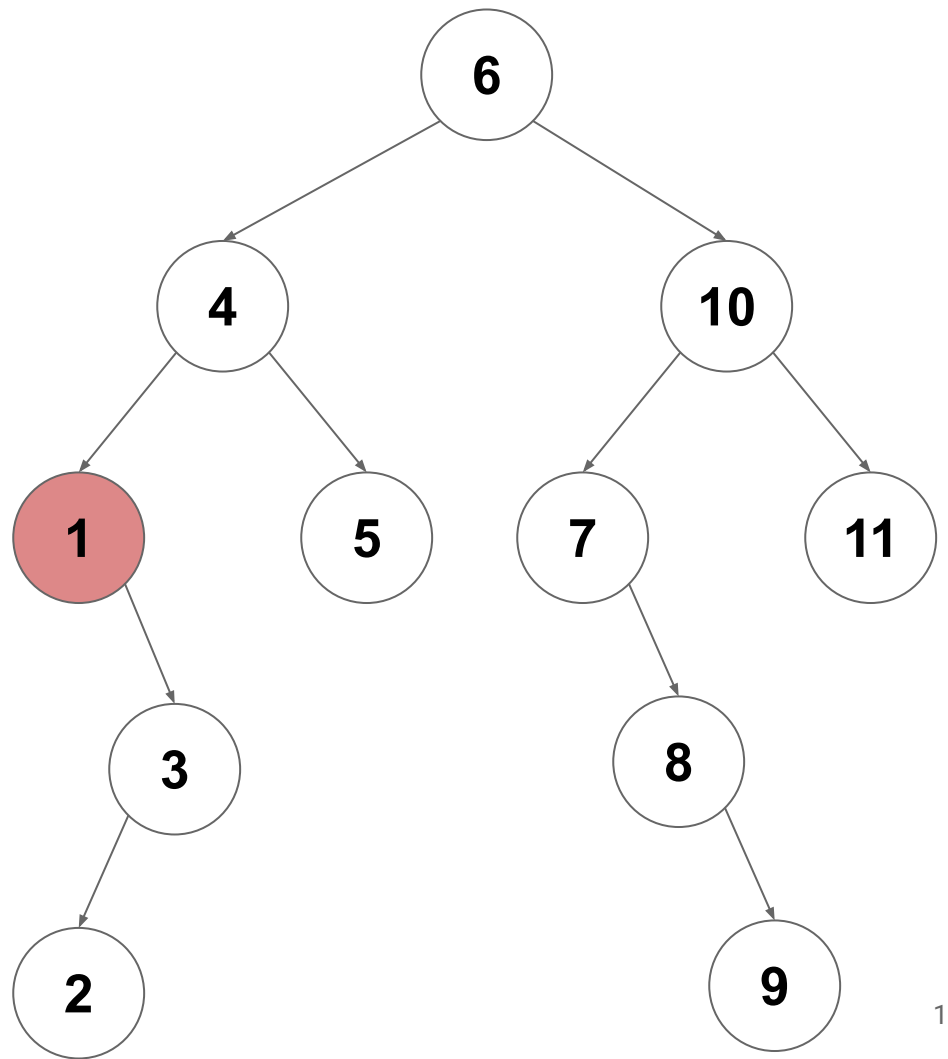


# In-Order Traversal on a BST

`inorderVisit(6)`

`inorderVisit(4)`

`inorderVisit(1)`



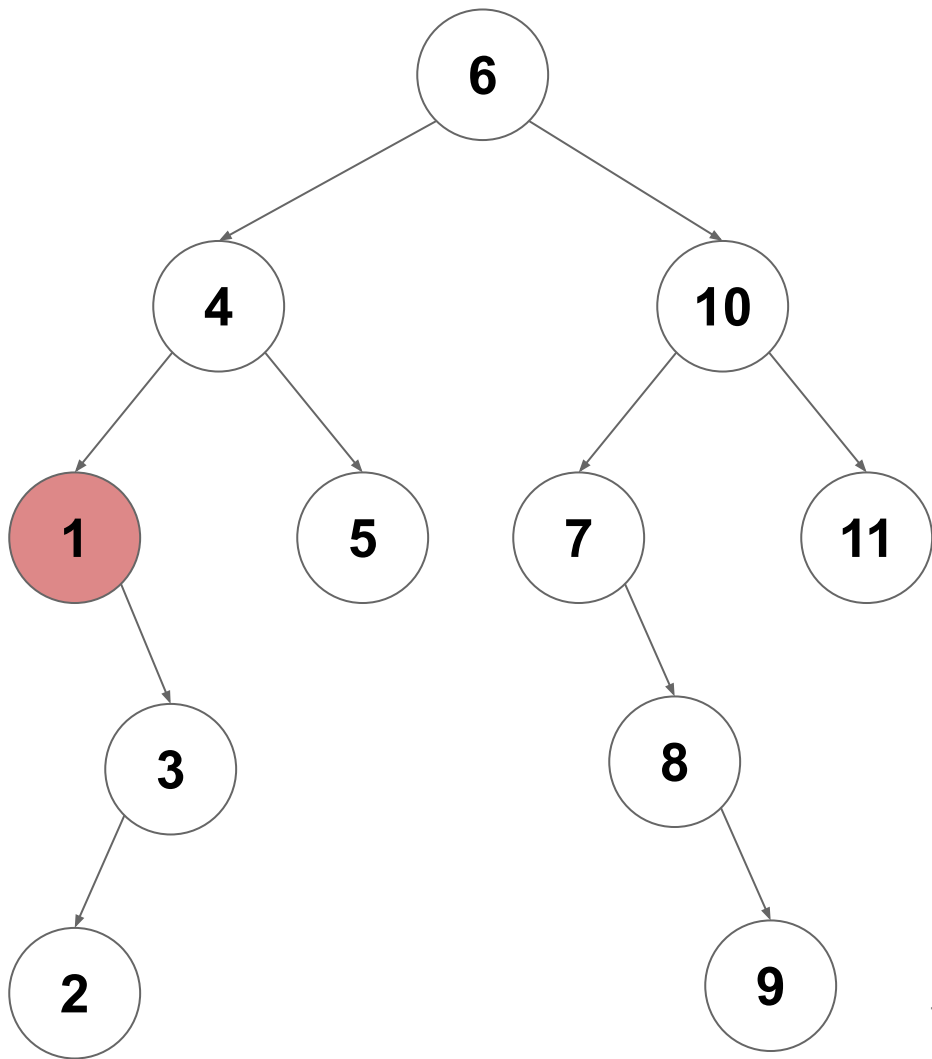
# In-Order Traversal on a BST

`inorderVisit(6)`

`inorderVisit(4)`

`inorderVisit(1)`

`inorderVisit(empty)`



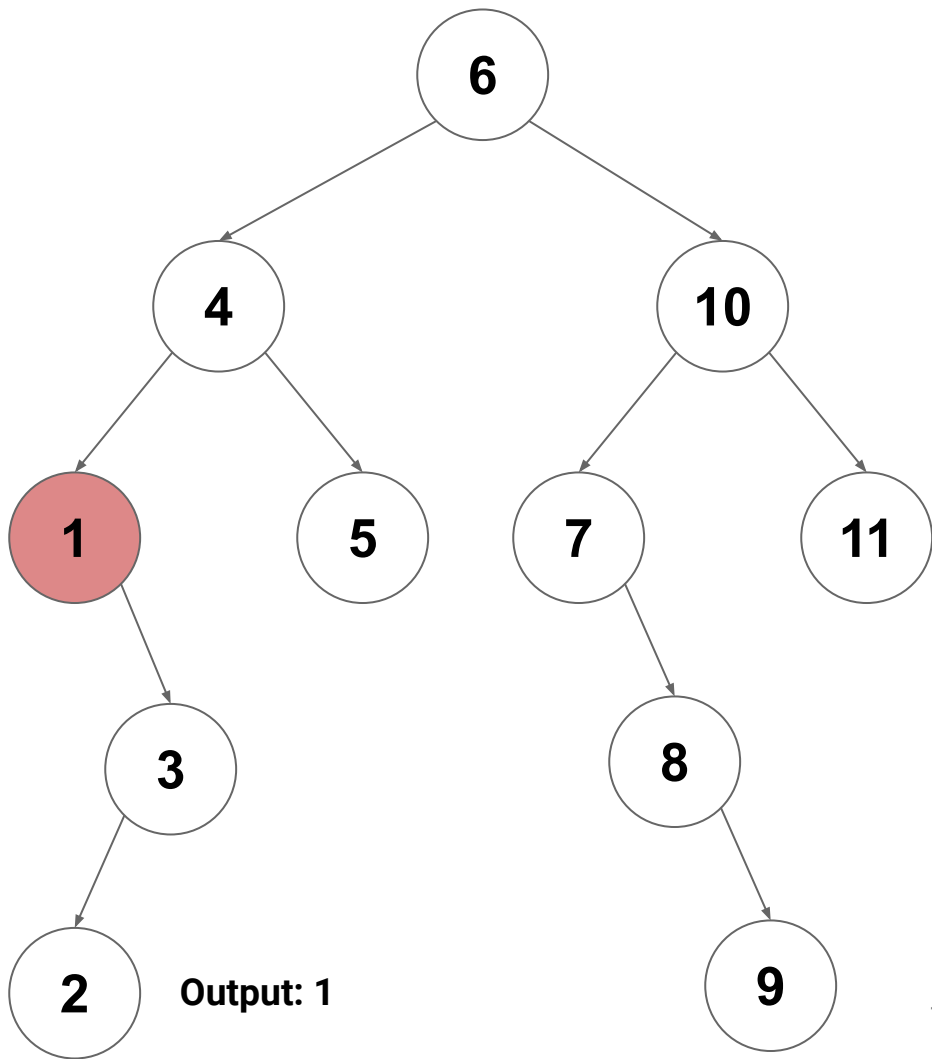
# In-Order Traversal on a BST

`inorderVisit(6)`

`inorderVisit(4)`

`inorderVisit(1)`

`visit(1)`



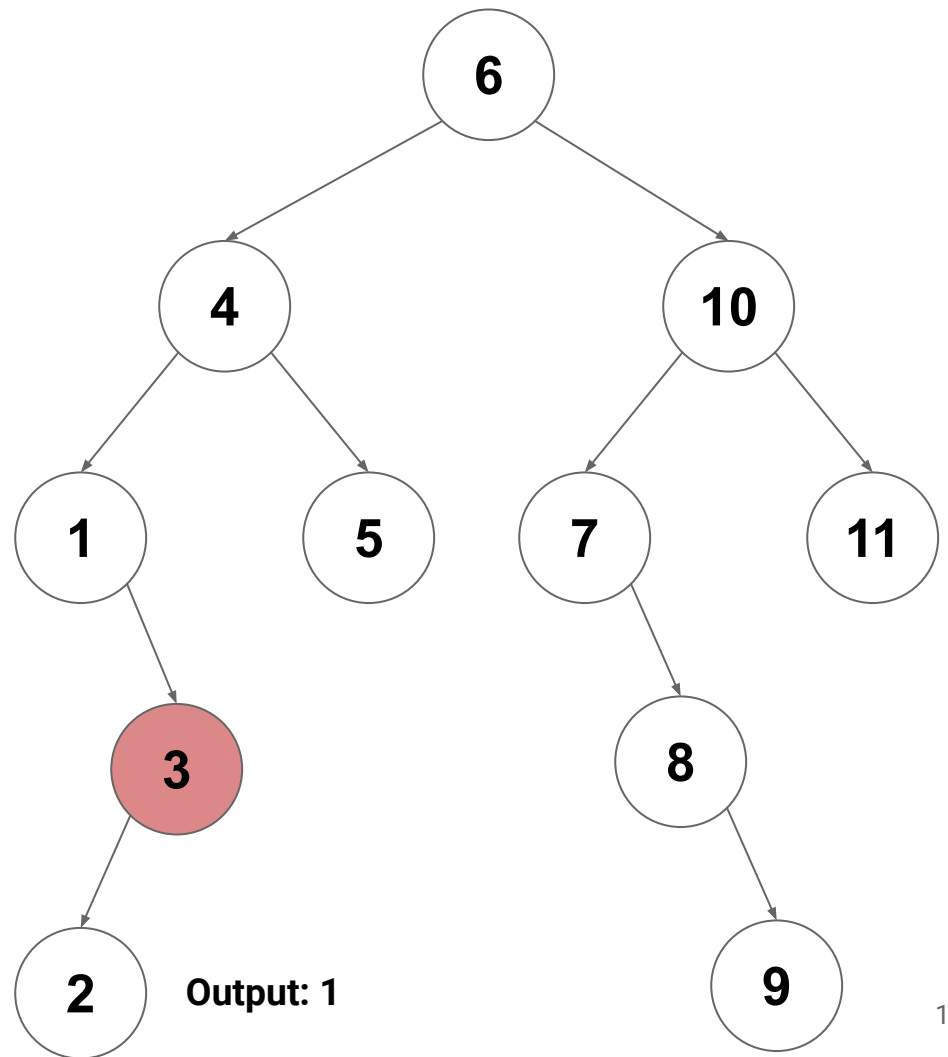
# In-Order Traversal on a BST

`inorderVisit(6)`

`inorderVisit(4)`

`inorderVisit(1)`

`inorderVisit(3)`



# In-Order Traversal on a BST

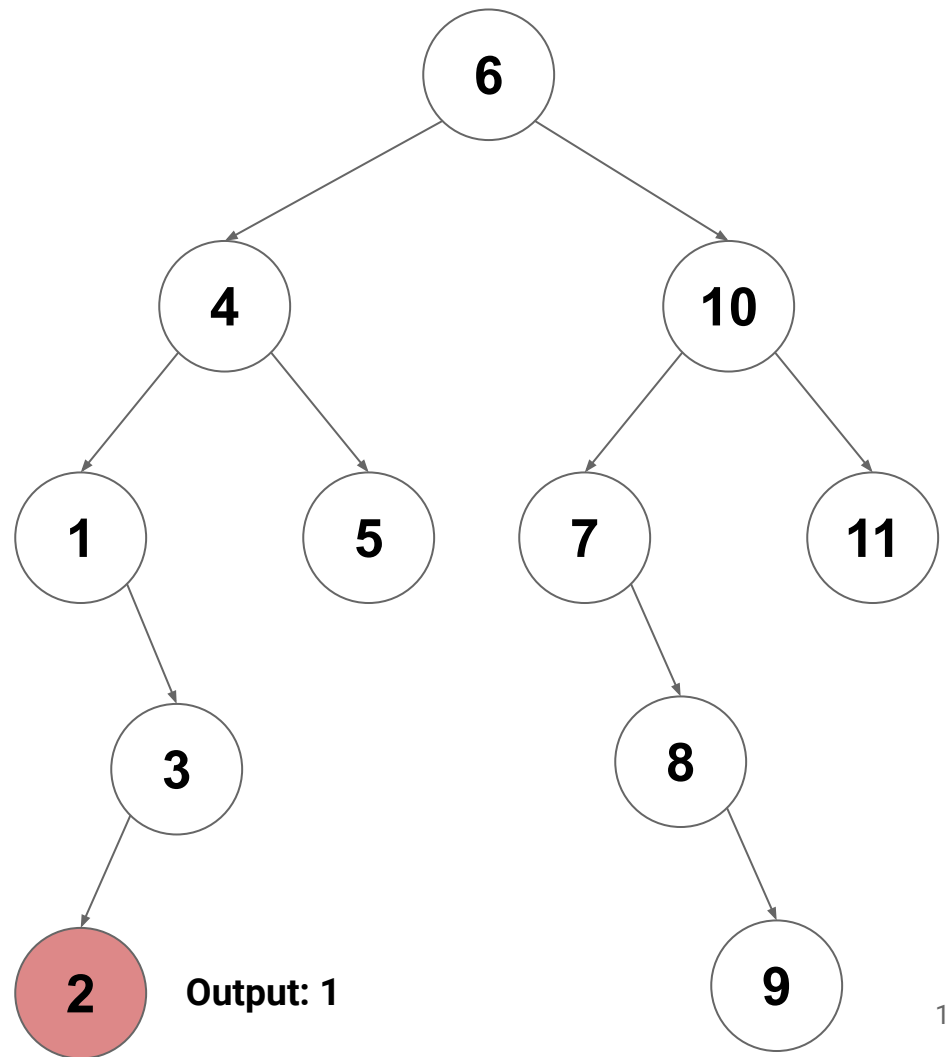
`inorderVisit(6)`

`inorderVisit(4)`

`inorderVisit(1)`

`inorderVisit(3)`

`inorderVisit(2)`





# In-Order Traversal on a BST

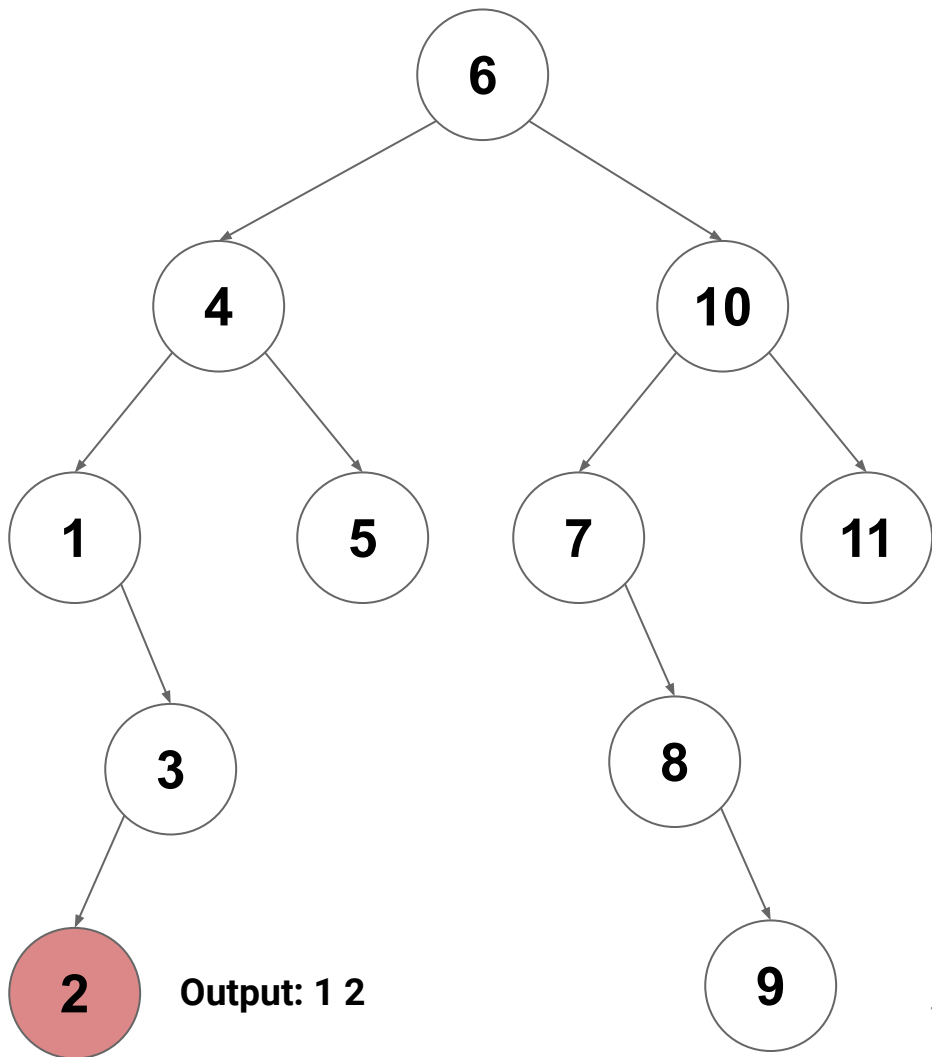
`inorderVisit(6)`

`inorderVisit(4)`

`inorderVisit(1)`

`inorderVisit(3)`

`visit(2)`



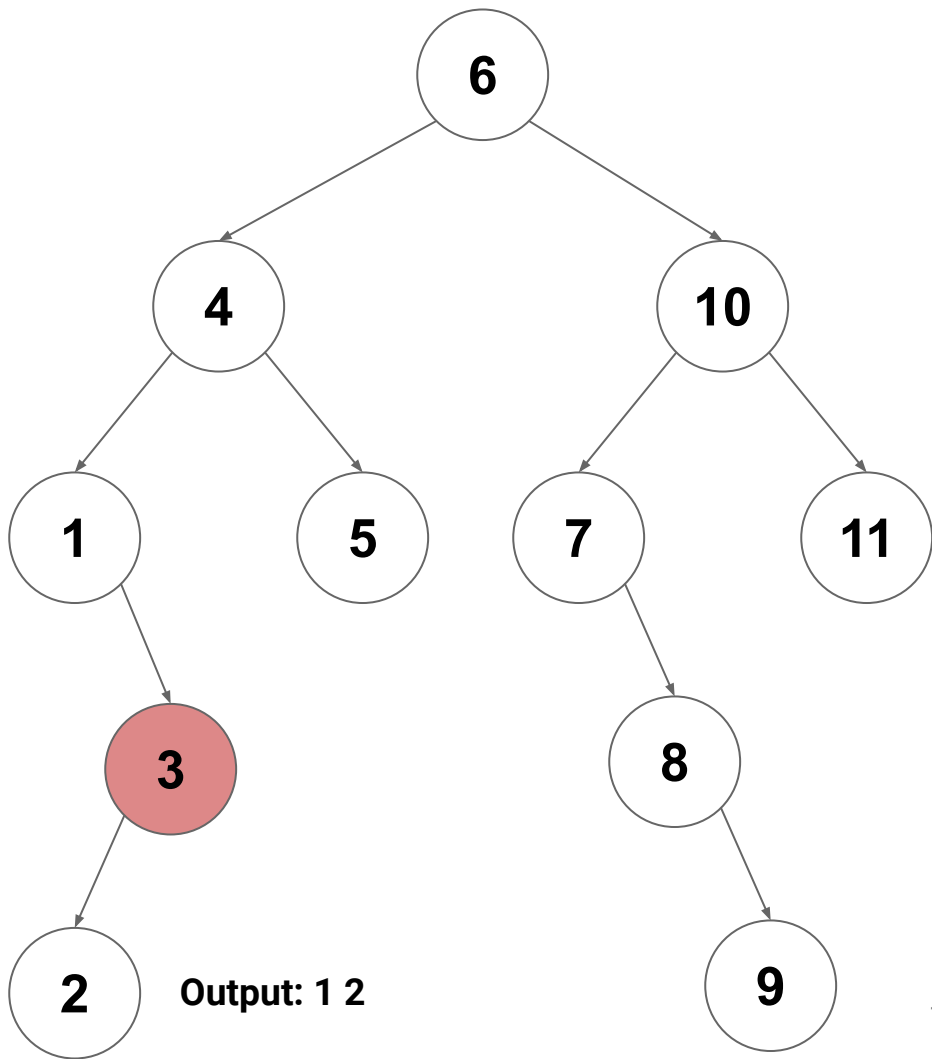
# In-Order Traversal on a BST

`inorderVisit(6)`

`inorderVisit(4)`

`inorderVisit(1)`

`inorderVisit(3)`



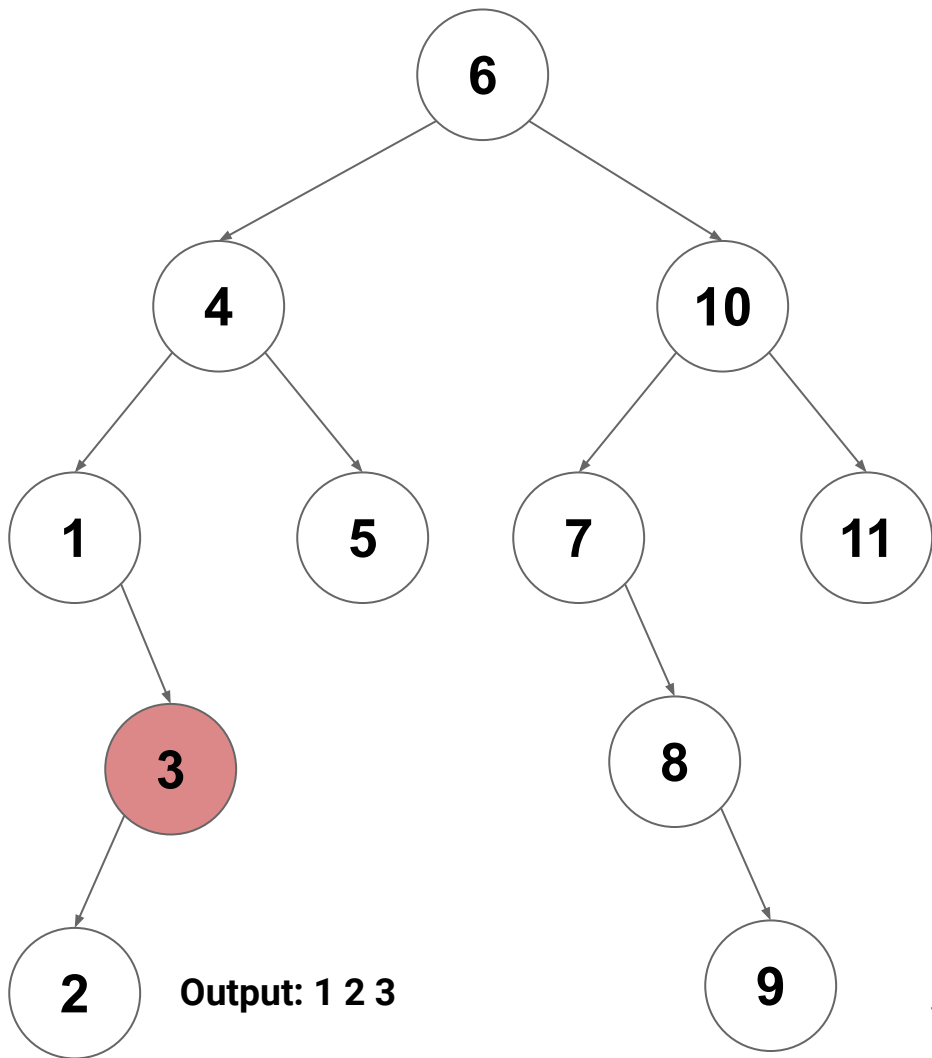
# In-Order Traversal on a BST

`inorderVisit(6)`

`inorderVisit(4)`

`inorderVisit(1)`

`visit(3)`

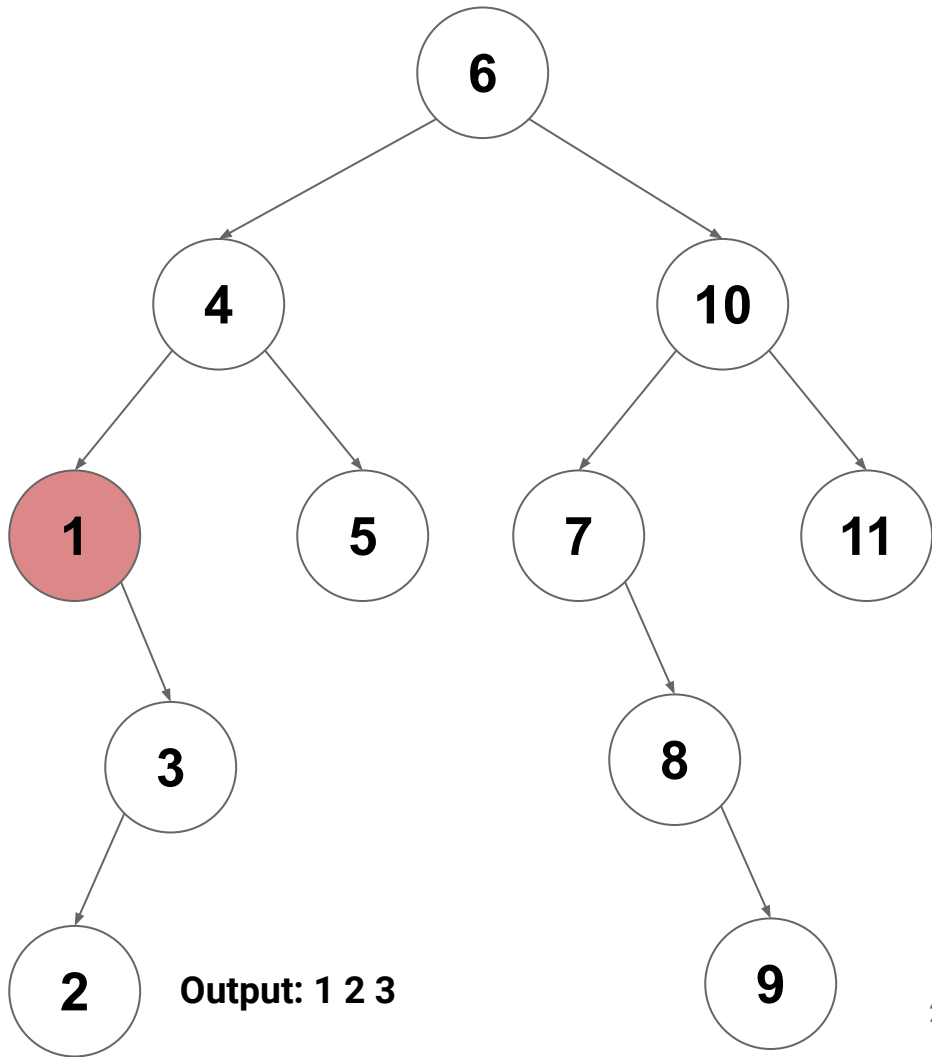


# In-Order Traversal on a BST

`inorderVisit(6)`

`inorderVisit(4)`

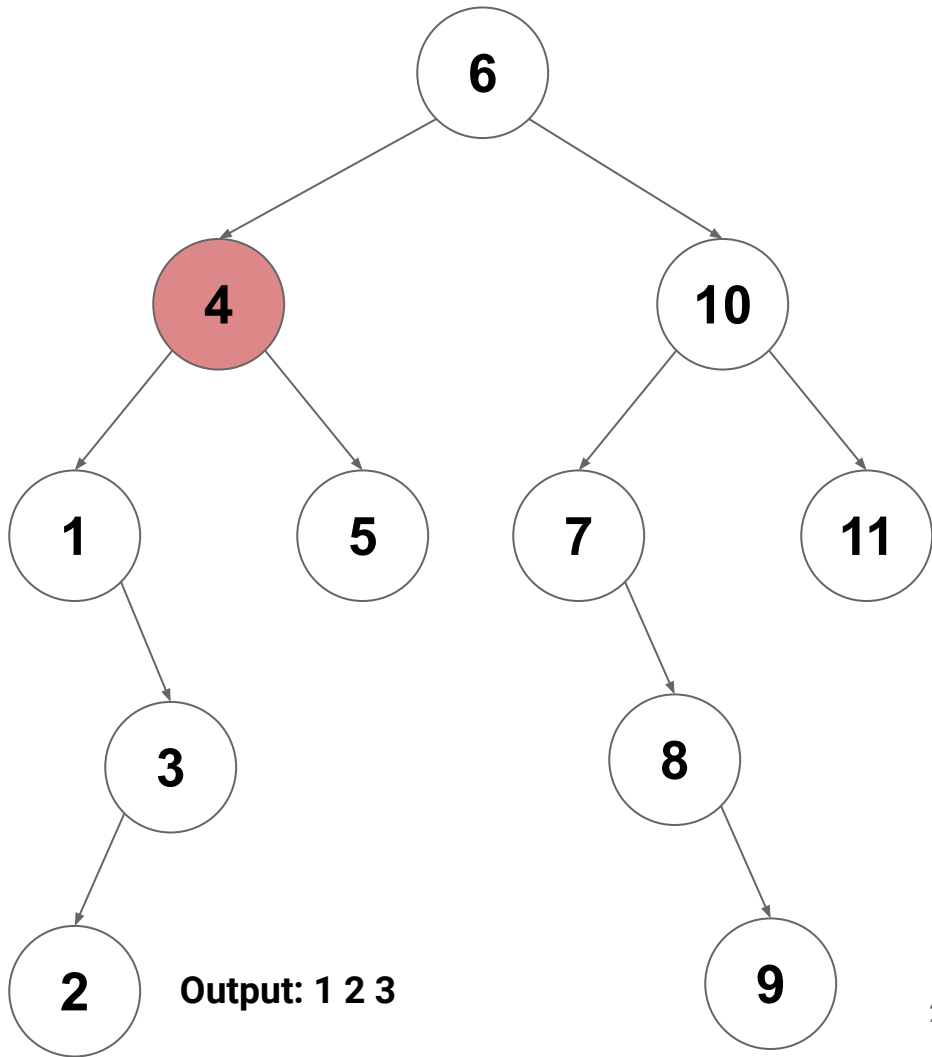
`inorderVisit(1)`



# In-Order Traversal on a BST

`inorderVisit(6)`

`inorderVisit(4)`

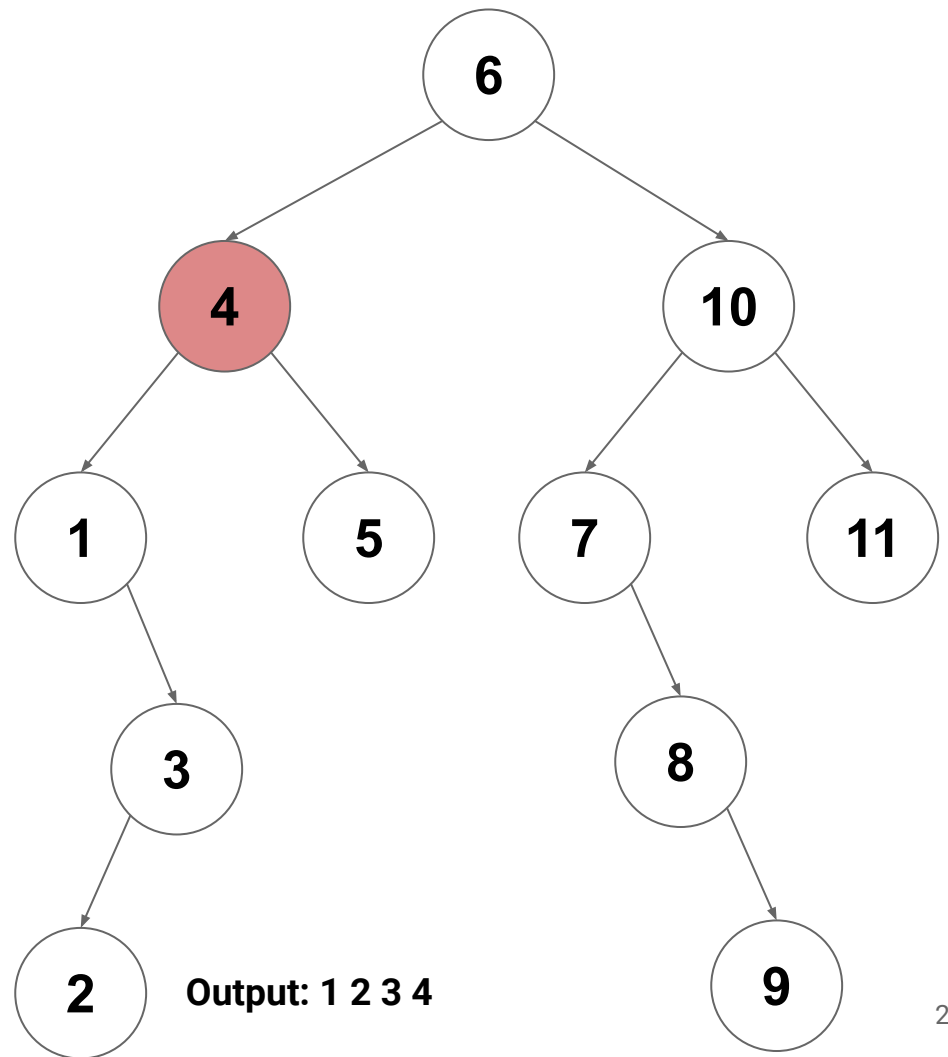


# In-Order Traversal on a BST

`inorderVisit(6)`

`inorderVisit(4)`

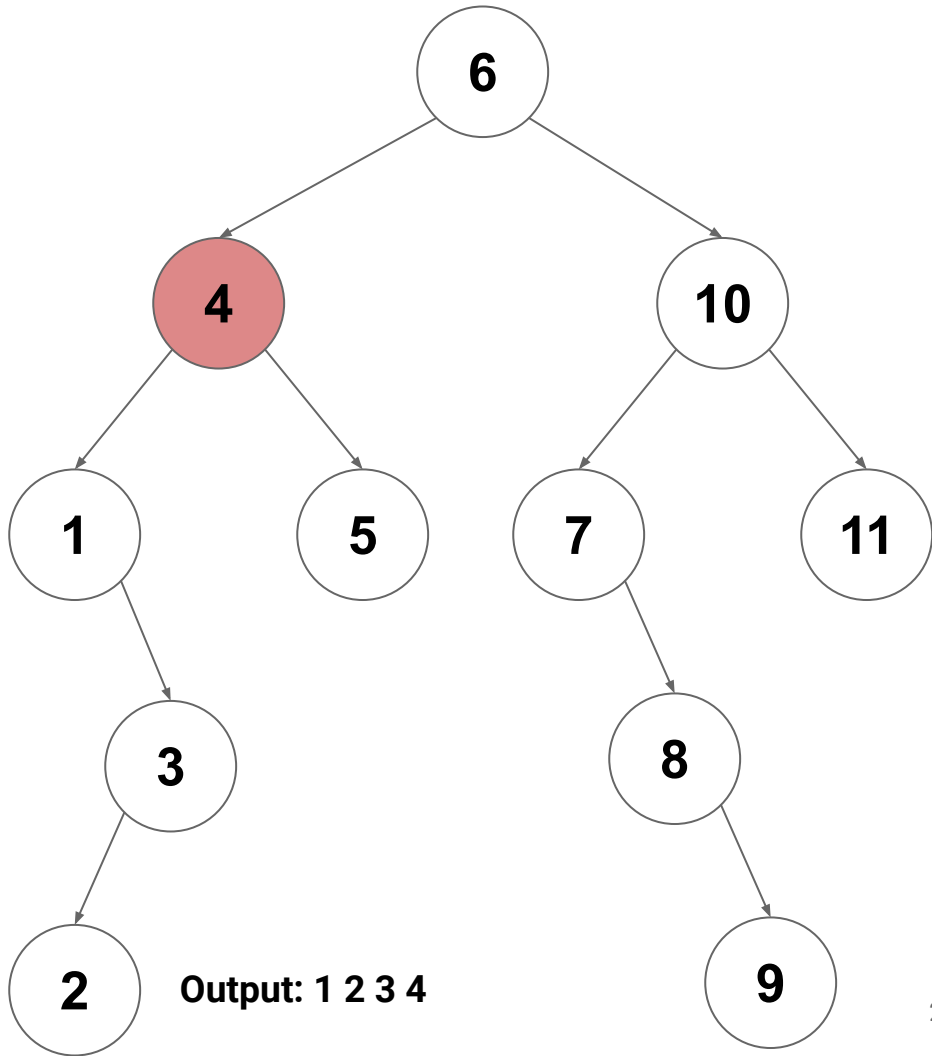
`visit(4)`



# In-Order Traversal on a BST

`inorderVisit(6)`

`inorderVisit(4)`

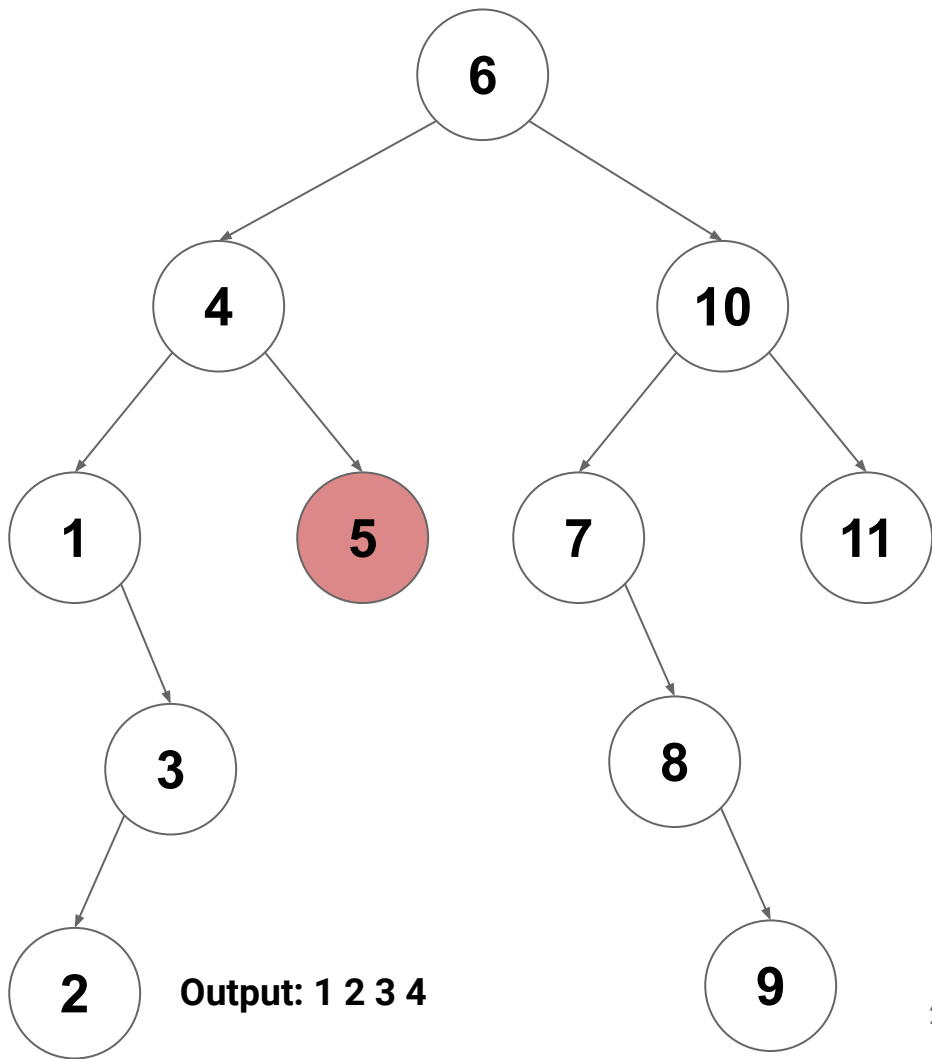


# In-Order Traversal on a BST

`inorderVisit(6)`

`inorderVisit(4)`

`inorderVisit(5)`



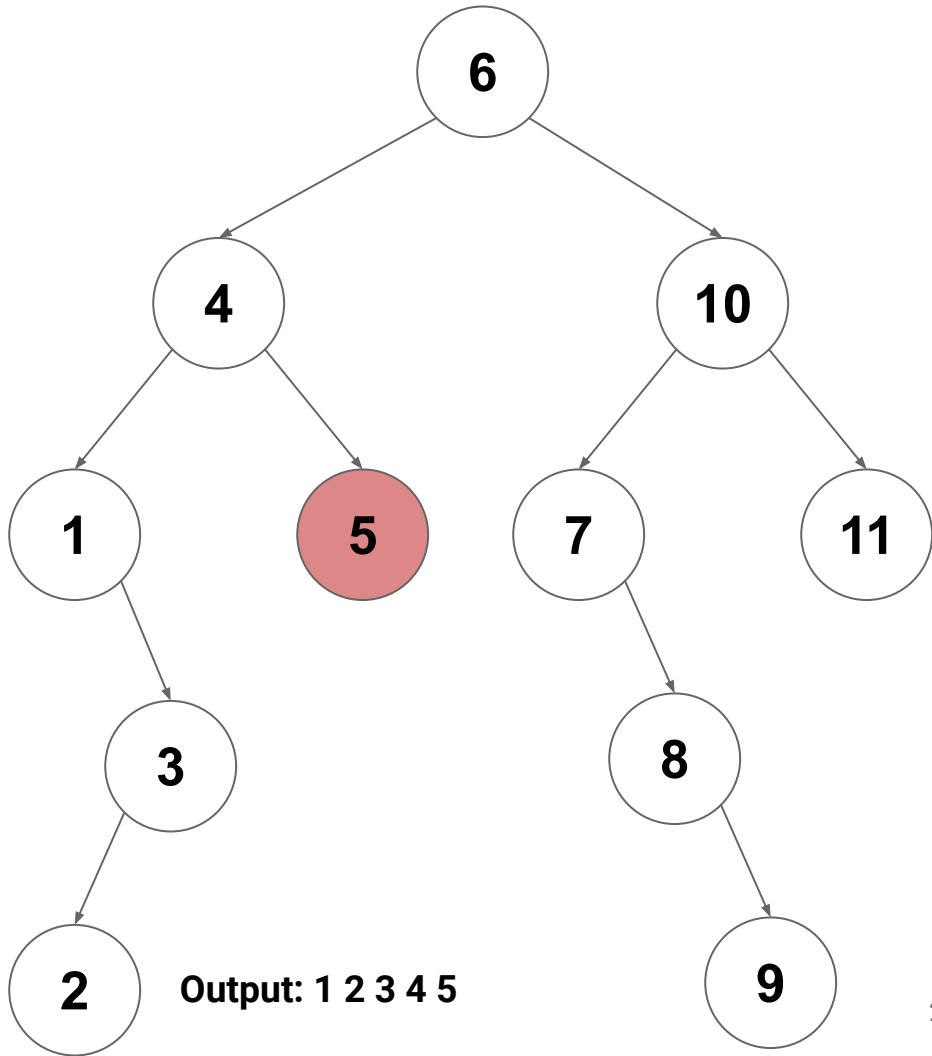


# In-Order Traversal on a BST

`inorderVisit(6)`

`inorderVisit(4)`

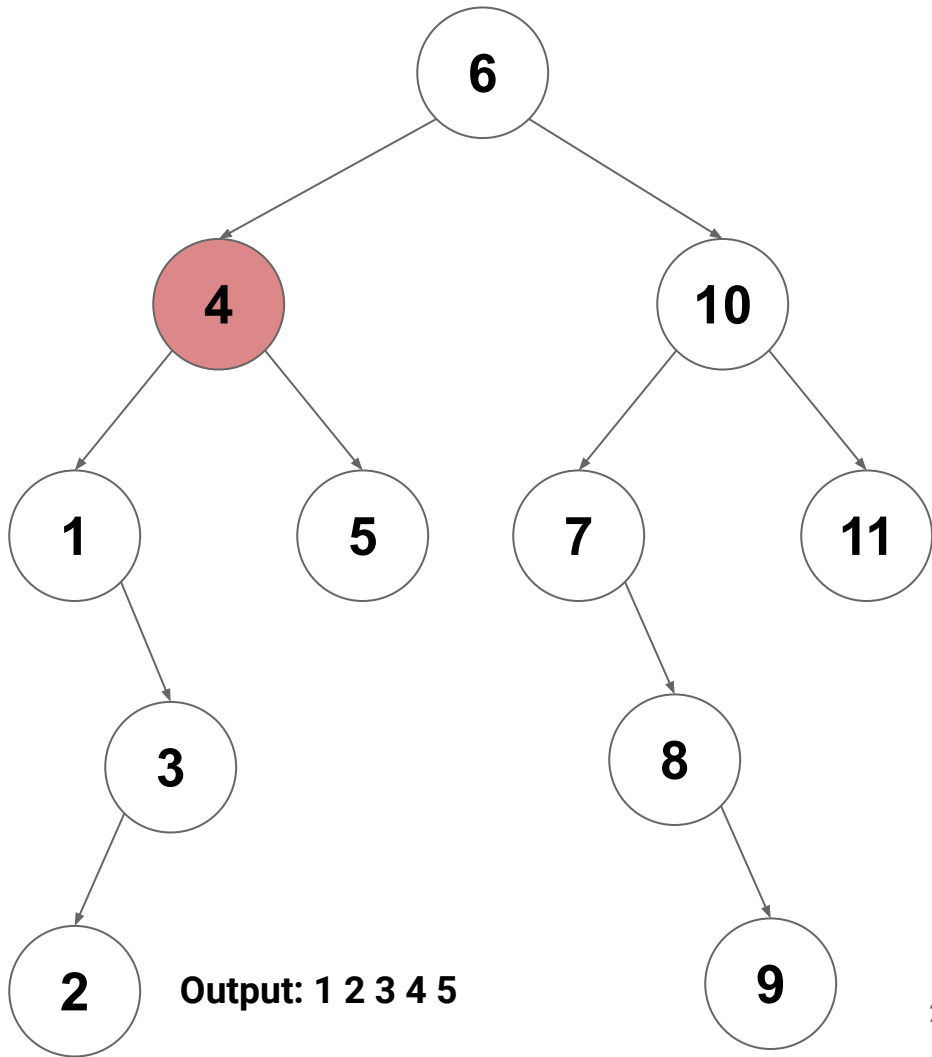
`visit(5)`



# In-Order Traversal on a BST

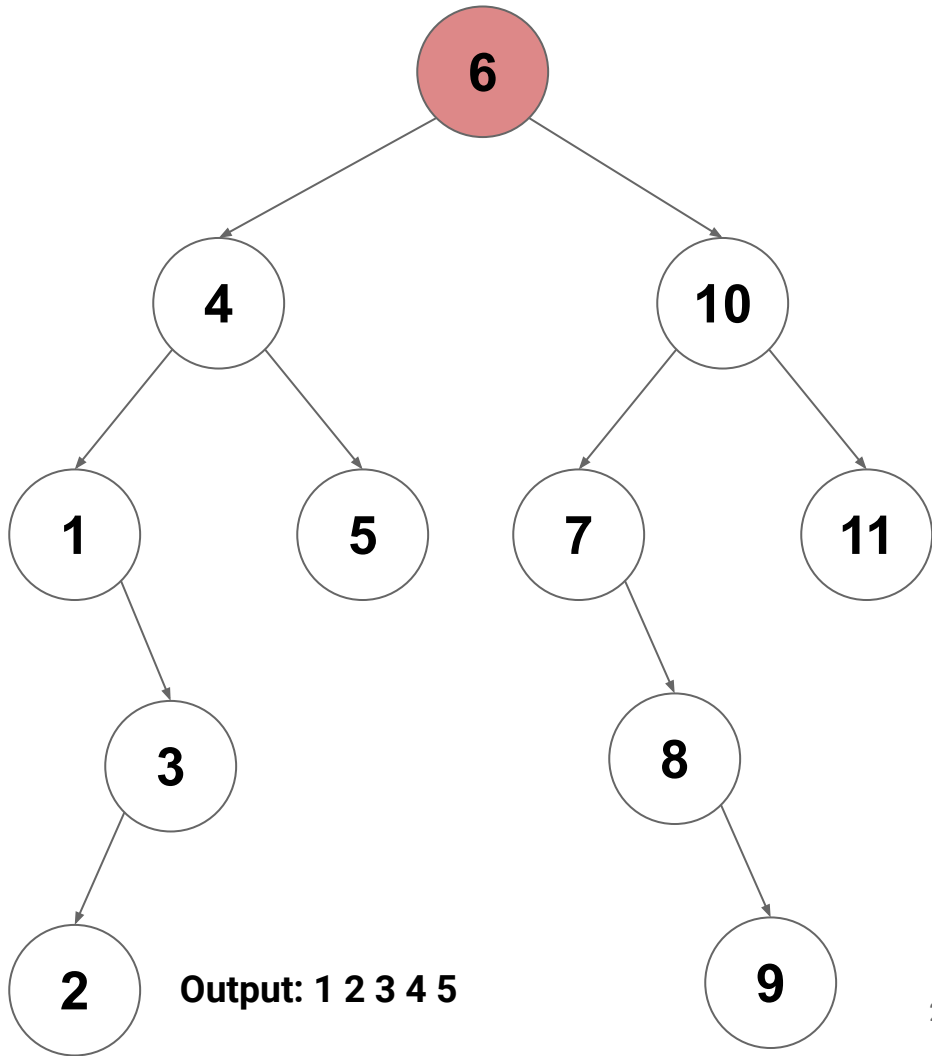
`inorderVisit(6)`

`inorderVisit(4)`



# In-Order Traversal on a BST

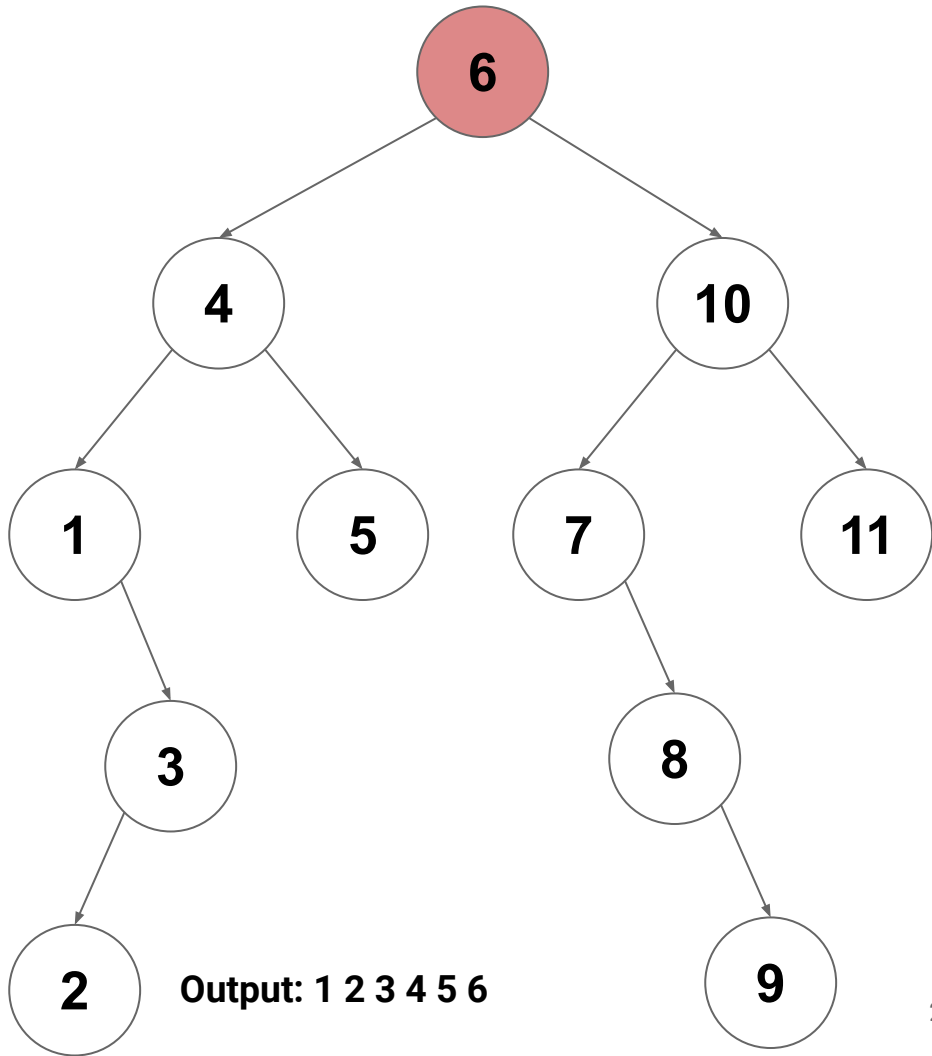
`inorderVisit(6)`



# In-Order Traversal on a BST

`inorderVisit(6)`

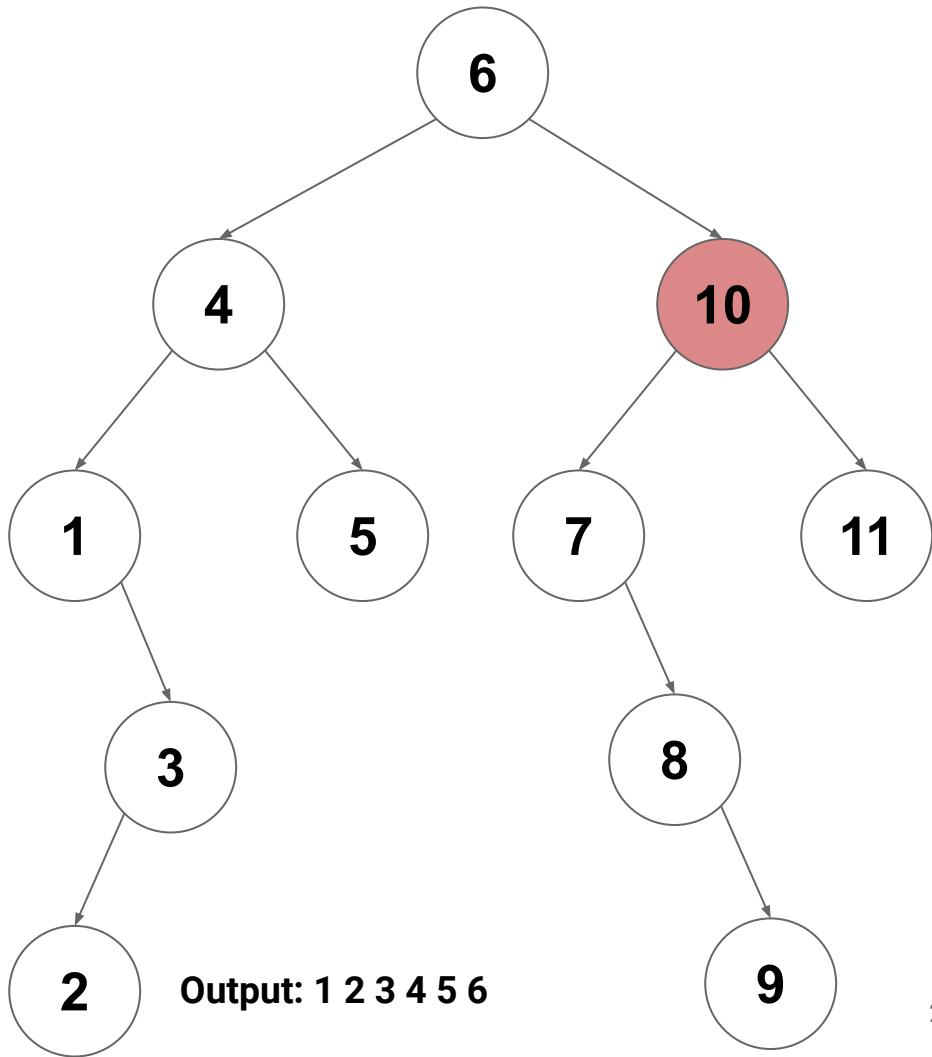
`visit(6)`



# In-Order Traversal on a BST

`inorderVisit(6)`

`inorderVisit(10)`

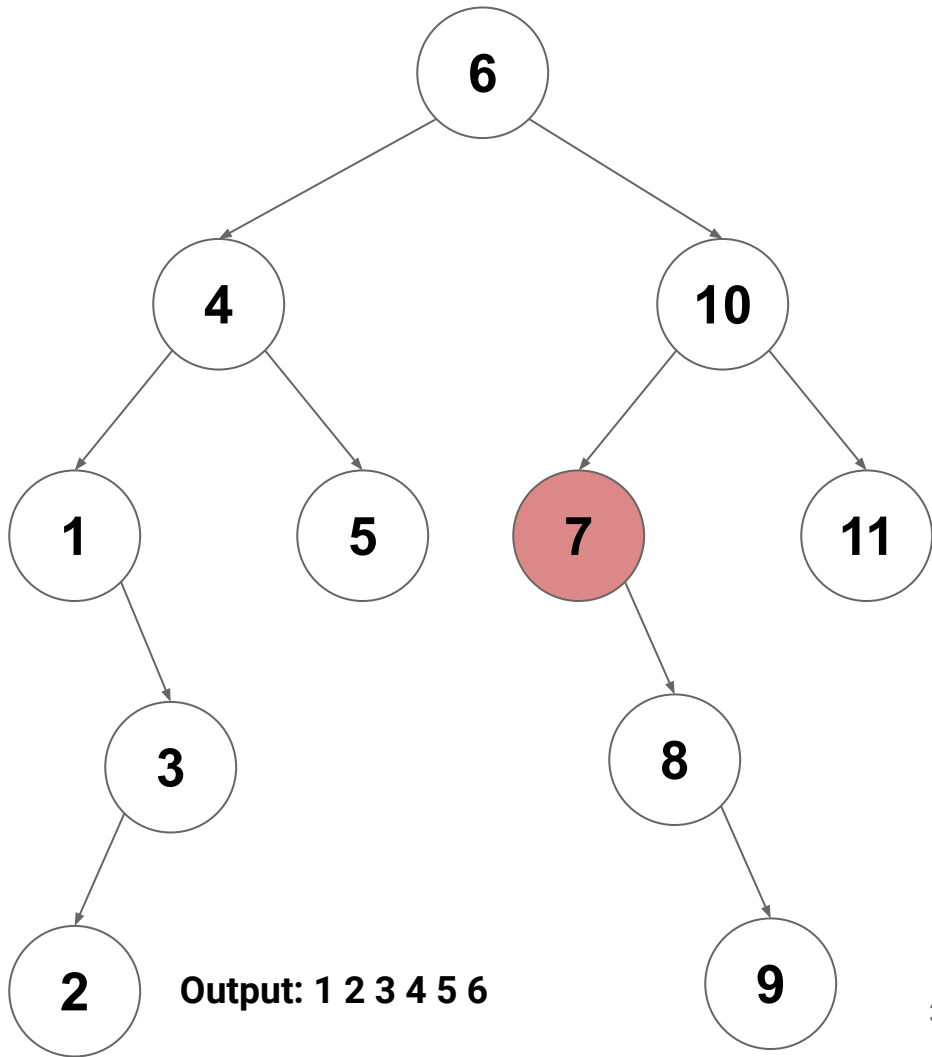


# In-Order Traversal on a BST

`inorderVisit(6)`

`inorderVisit(10)`

`inorderVisit(7)`



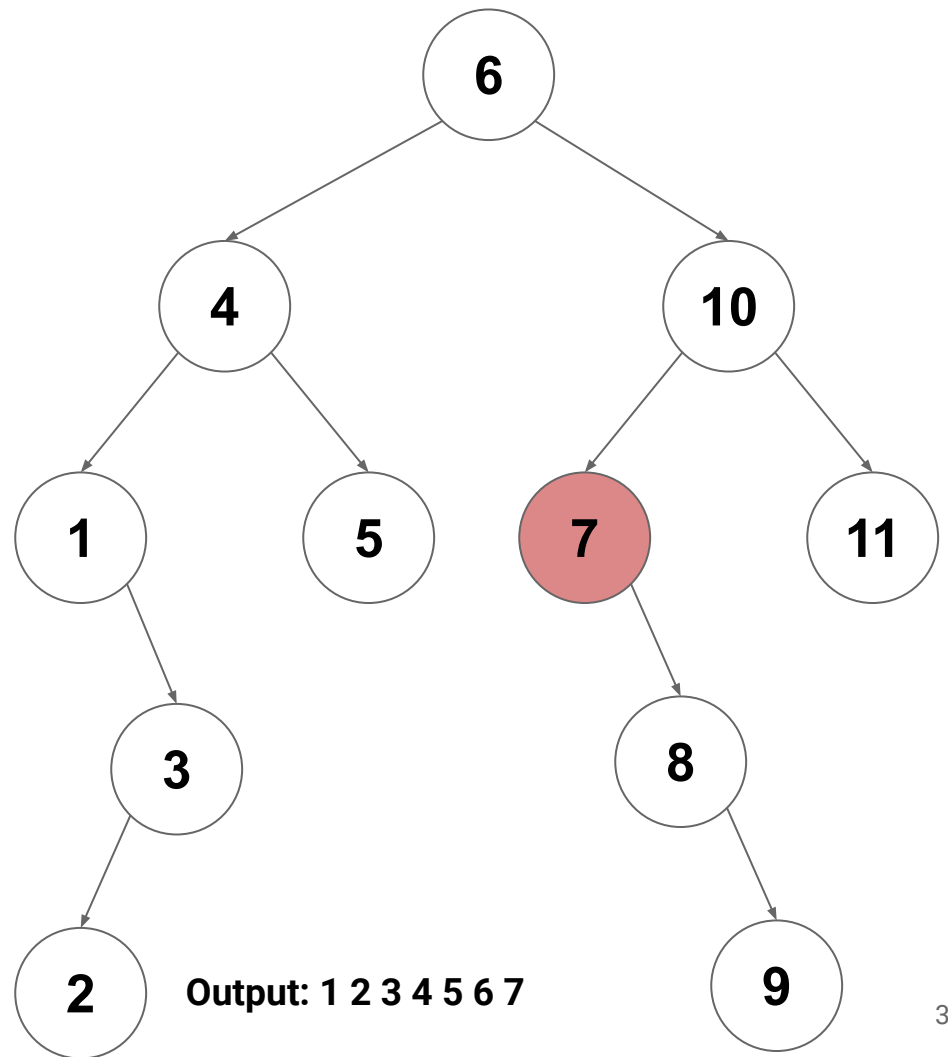
# In-Order Traversal on a BST

`inorderVisit(6)`

`inorderVisit(10)`

`inorderVisit(7)`

`visit(7)`



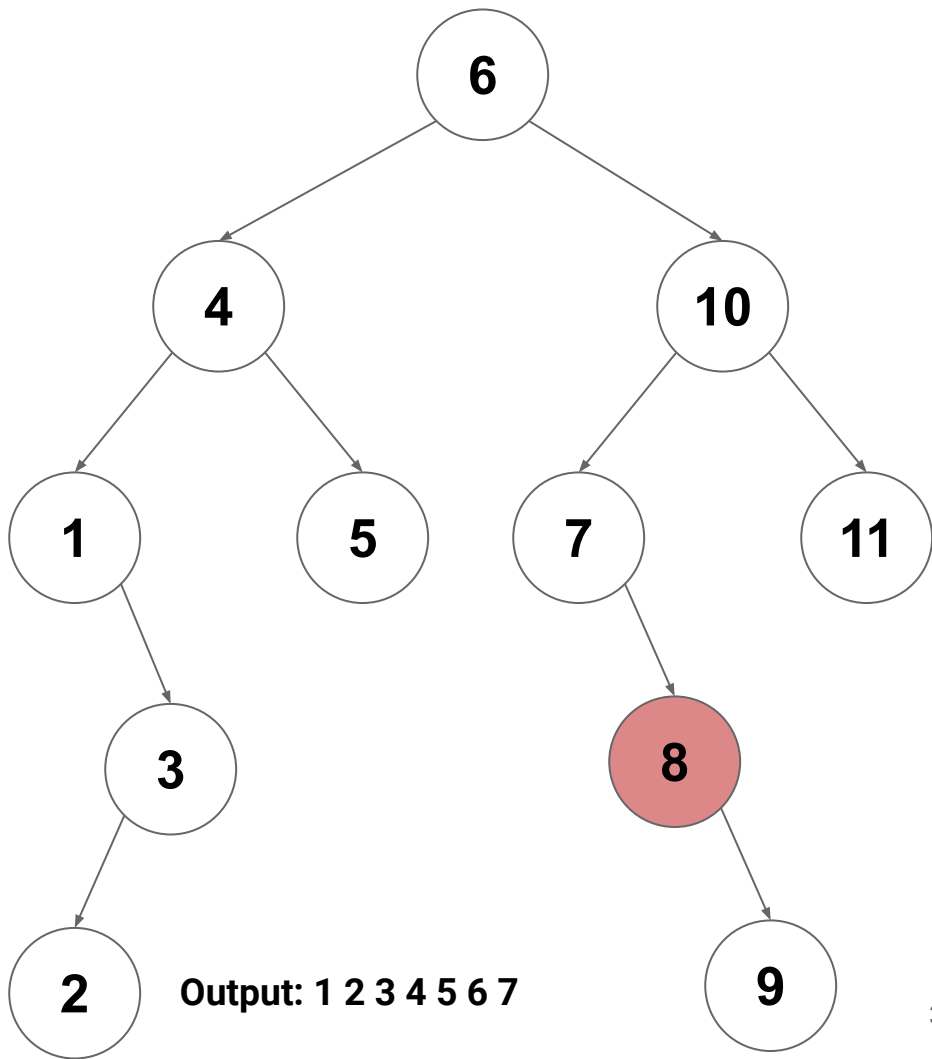
# In-Order Traversal on a BST

`inorderVisit(6)`

`inorderVisit(10)`

`inorderVisit(7)`

`inorderVisit(8)`





# In-Order Traversal on a BST

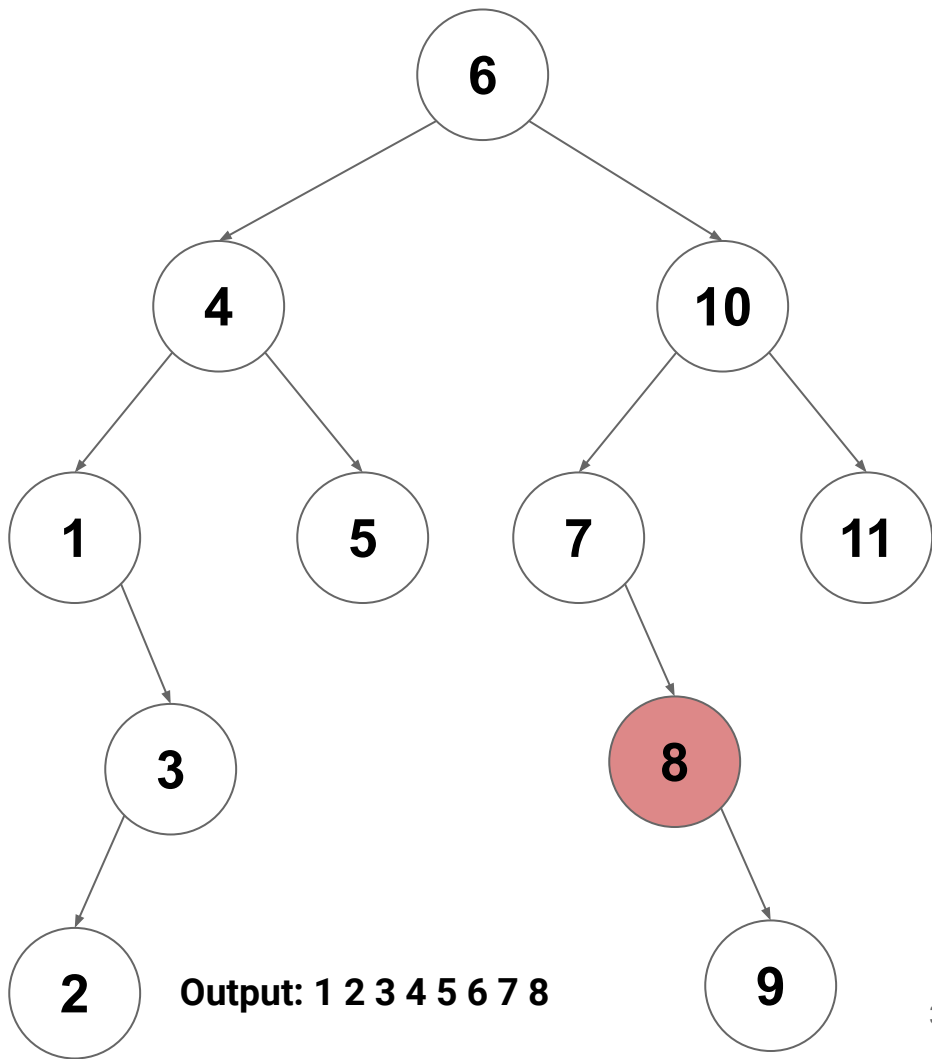
`inorderVisit(6)`

`inorderVisit(10)`

`inorderVisit(7)`

`inorderVisit(8)`

`visit(8)`



# In-Order Traversal on a BST

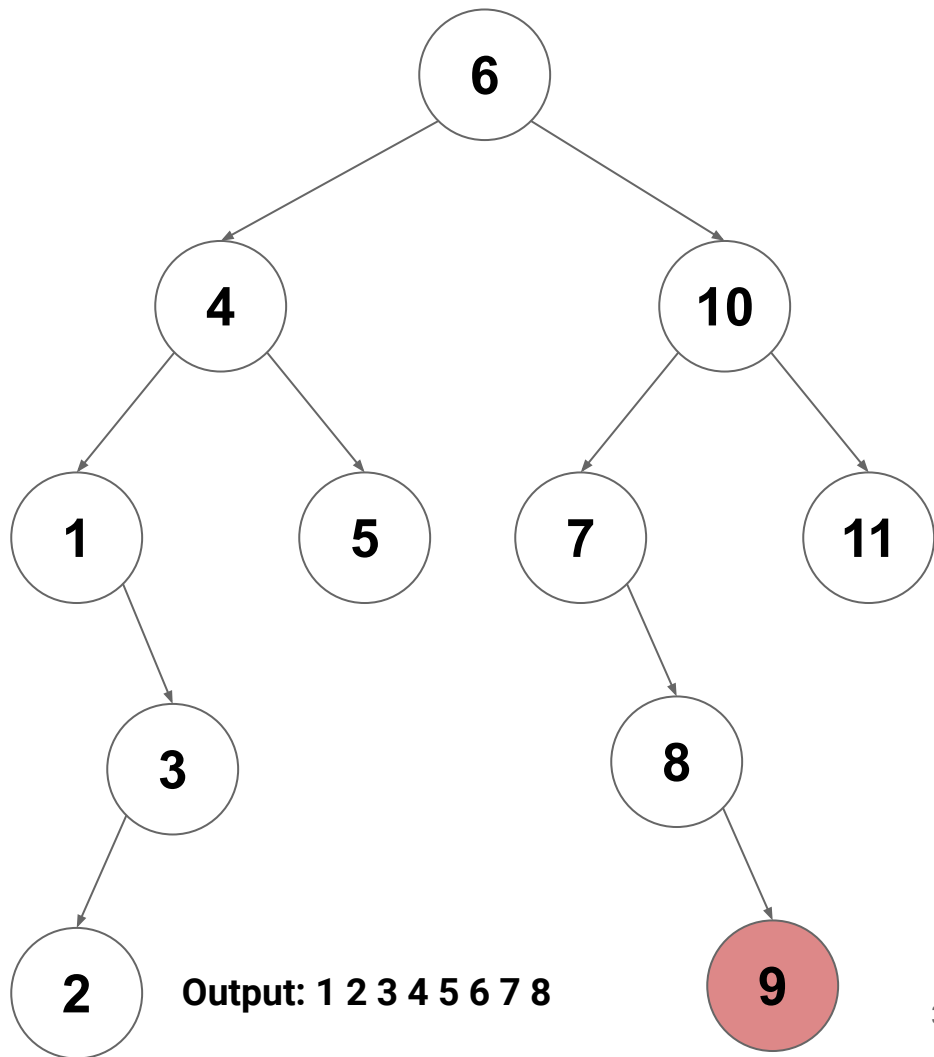
`inorderVisit(6)`

`inorderVisit(10)`

`inorderVisit(7)`

`inorderVisit(8)`

`inorderVisit(9)`



# In-Order Traversal on a BST

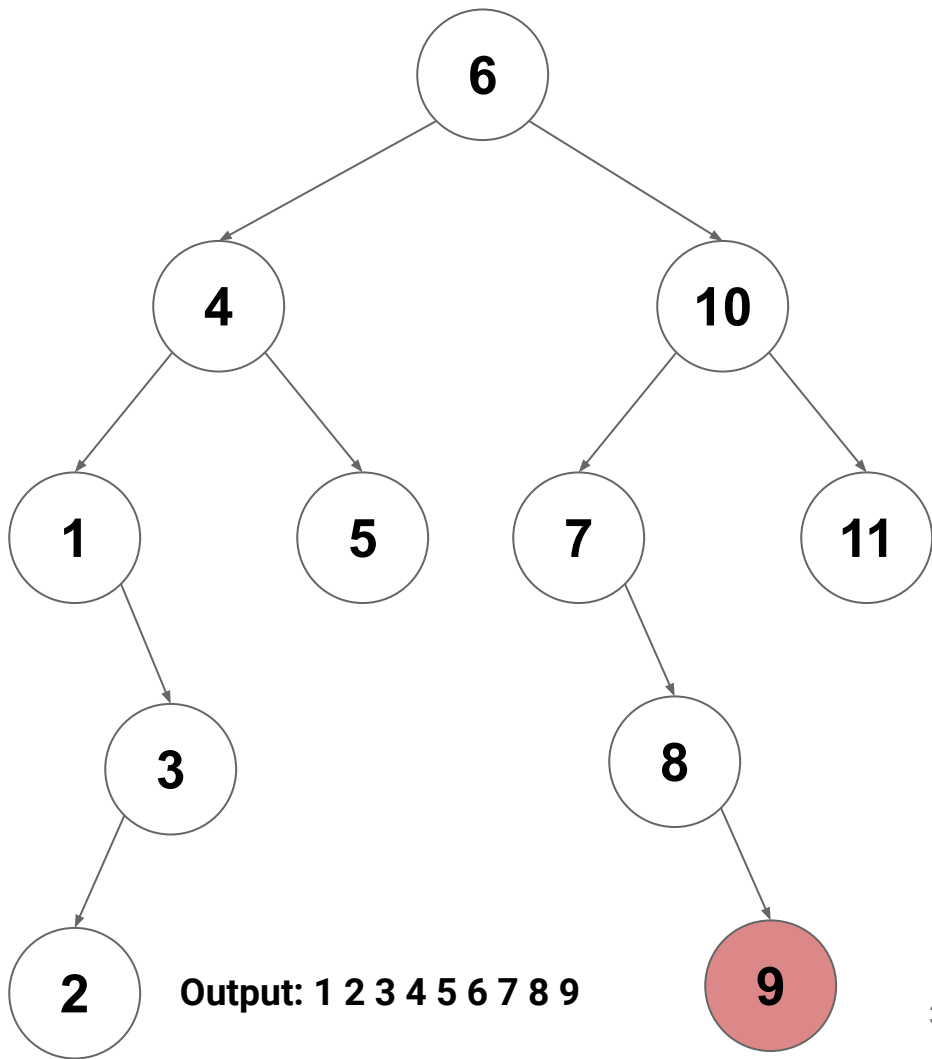
`inorderVisit(6)`

`inorderVisit(10)`

`inorderVisit(7)`

`inorderVisit(8)`

`visit(9)`



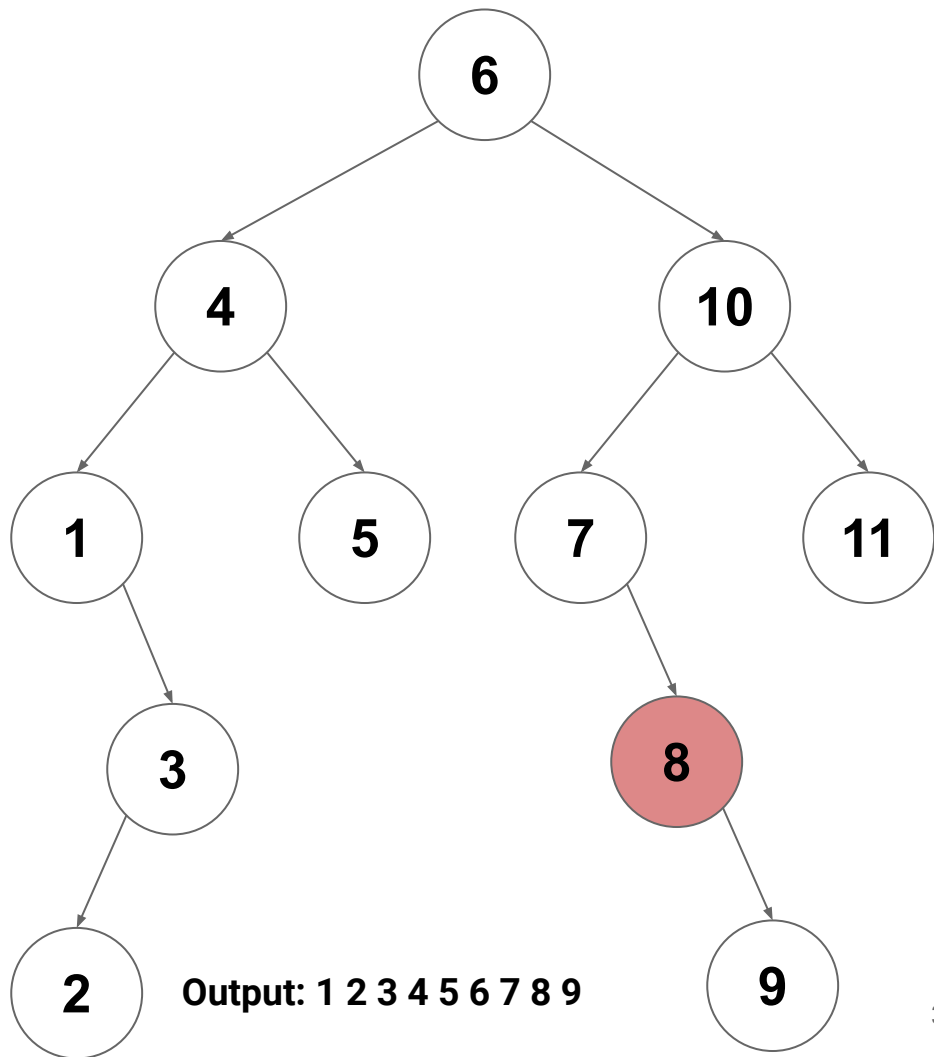
# In-Order Traversal on a BST

`inorderVisit(6)`

`inorderVisit(10)`

`inorderVisit(7)`

`inorderVisit(8)`

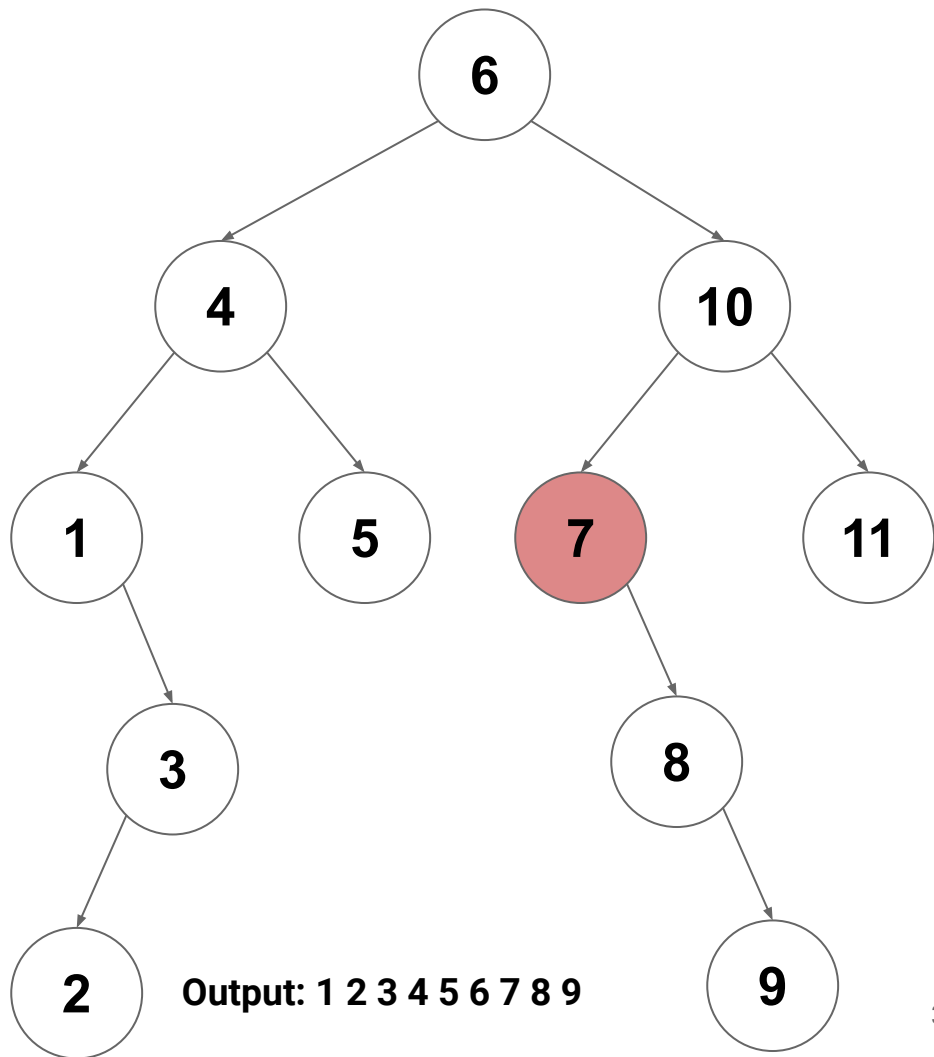


# In-Order Traversal on a BST

`inorderVisit(6)`

`inorderVisit(10)`

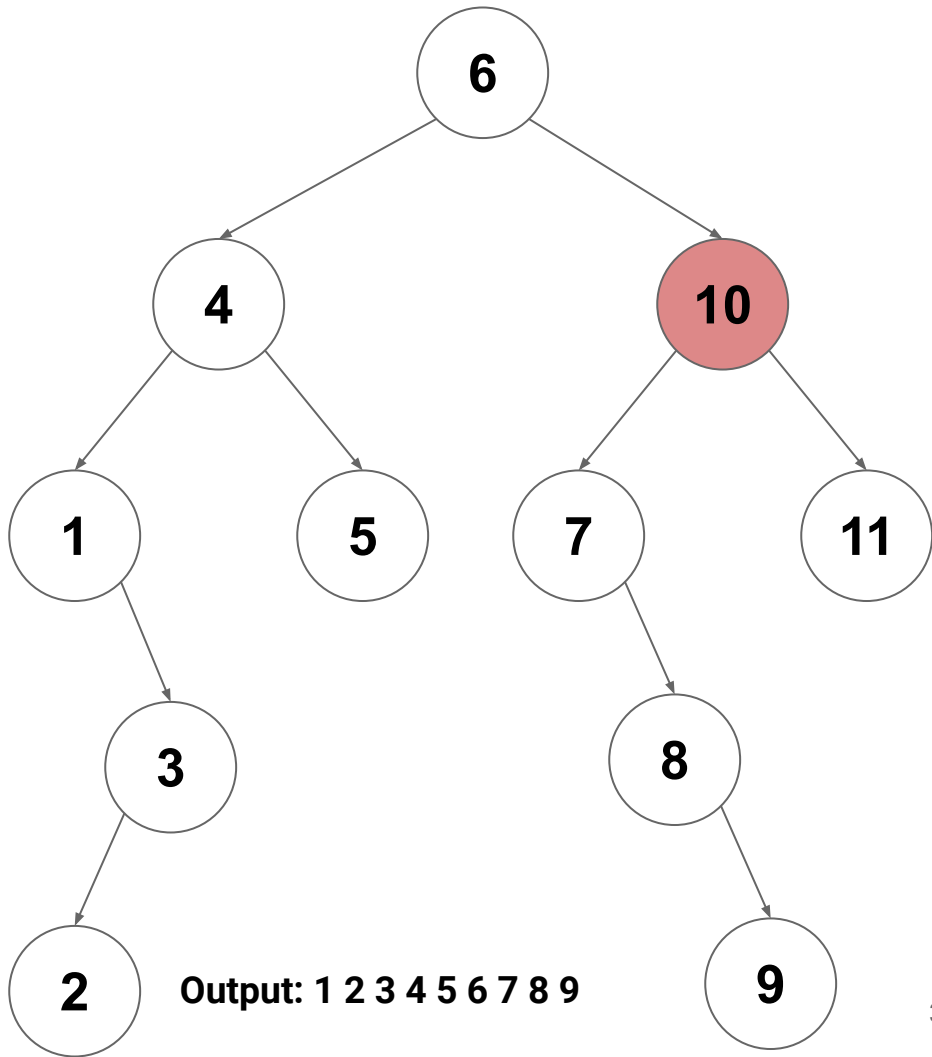
`inorderVisit(7)`



# In-Order Traversal on a BST

`inorderVisit(6)`

`inorderVisit(10)`

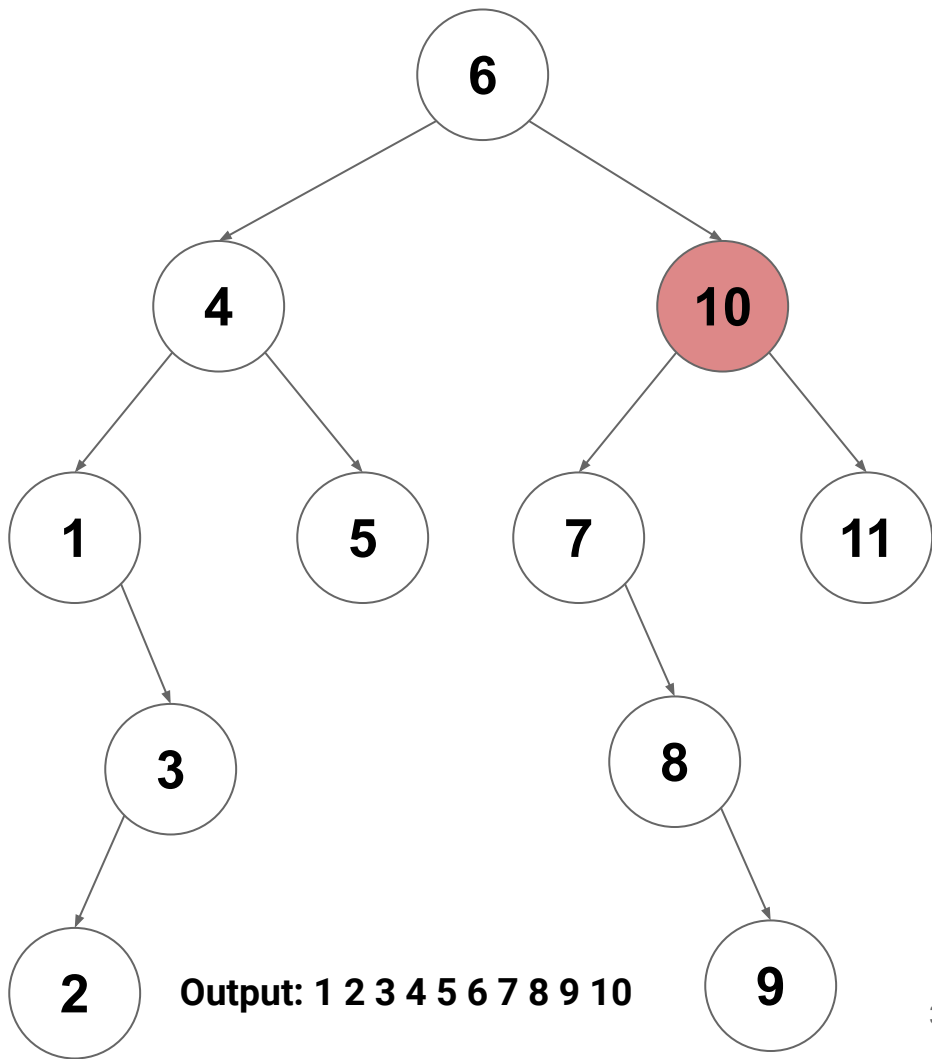


# In-Order Traversal on a BST

`inorderVisit(6)`

`inorderVisit(10)`

`visit(10)`

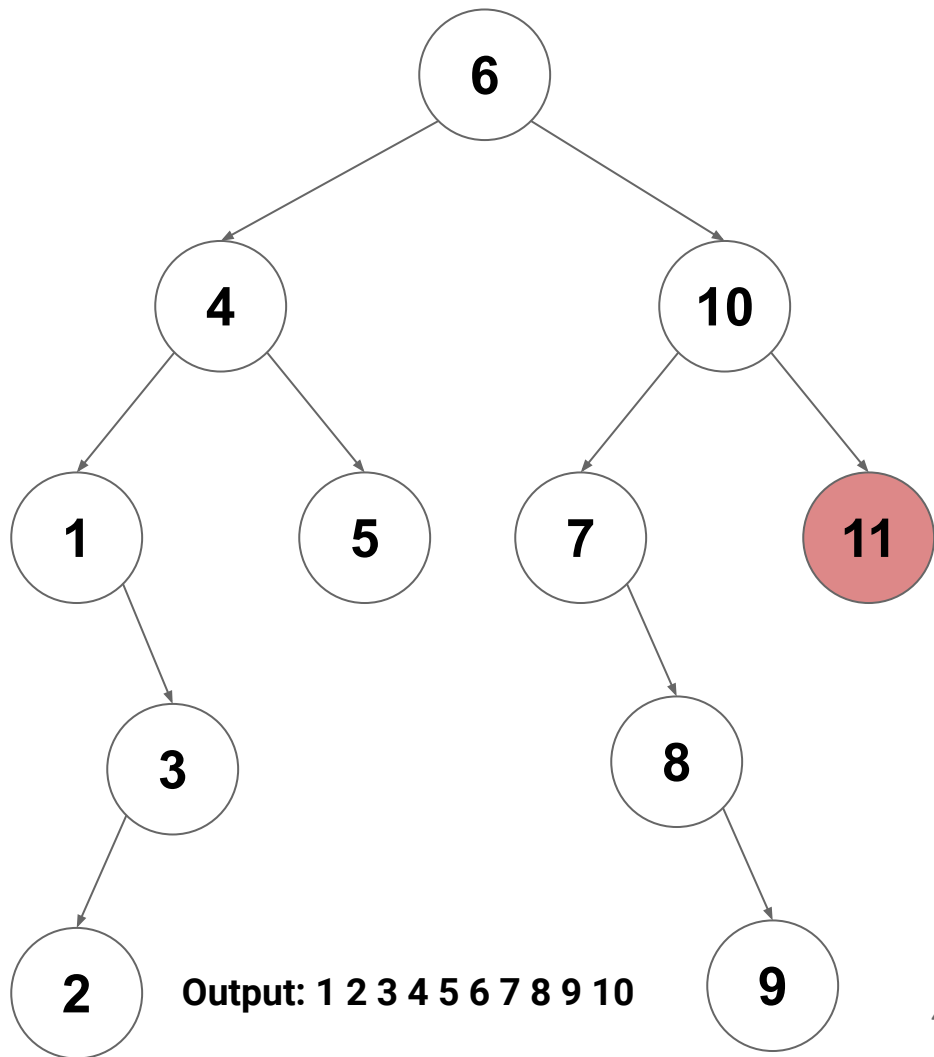


# In-Order Traversal on a BST

`inorderVisit(6)`

`inorderVisit(10)`

`inorderVisit(11)`





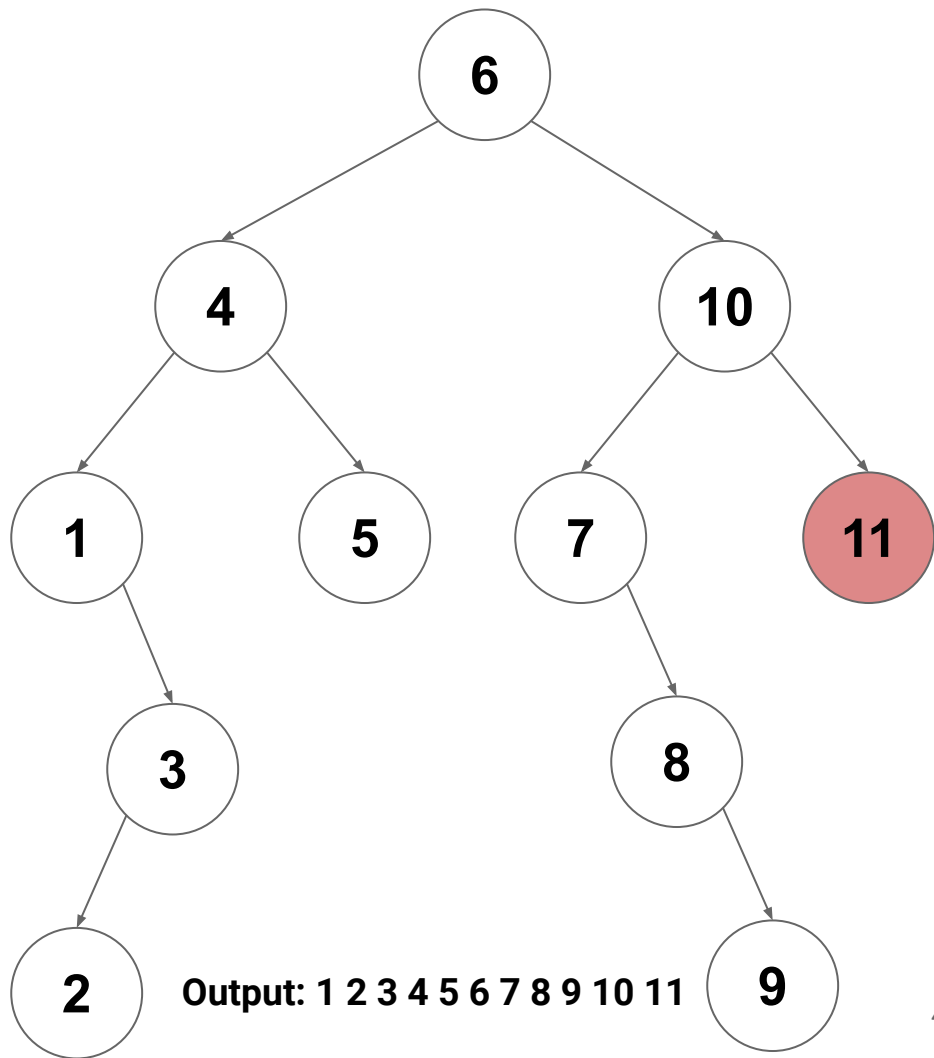
# In-Order Traversal on a BST

`inorderVisit(6)`

`inorderVisit(10)`

`inorderVisit(11)`

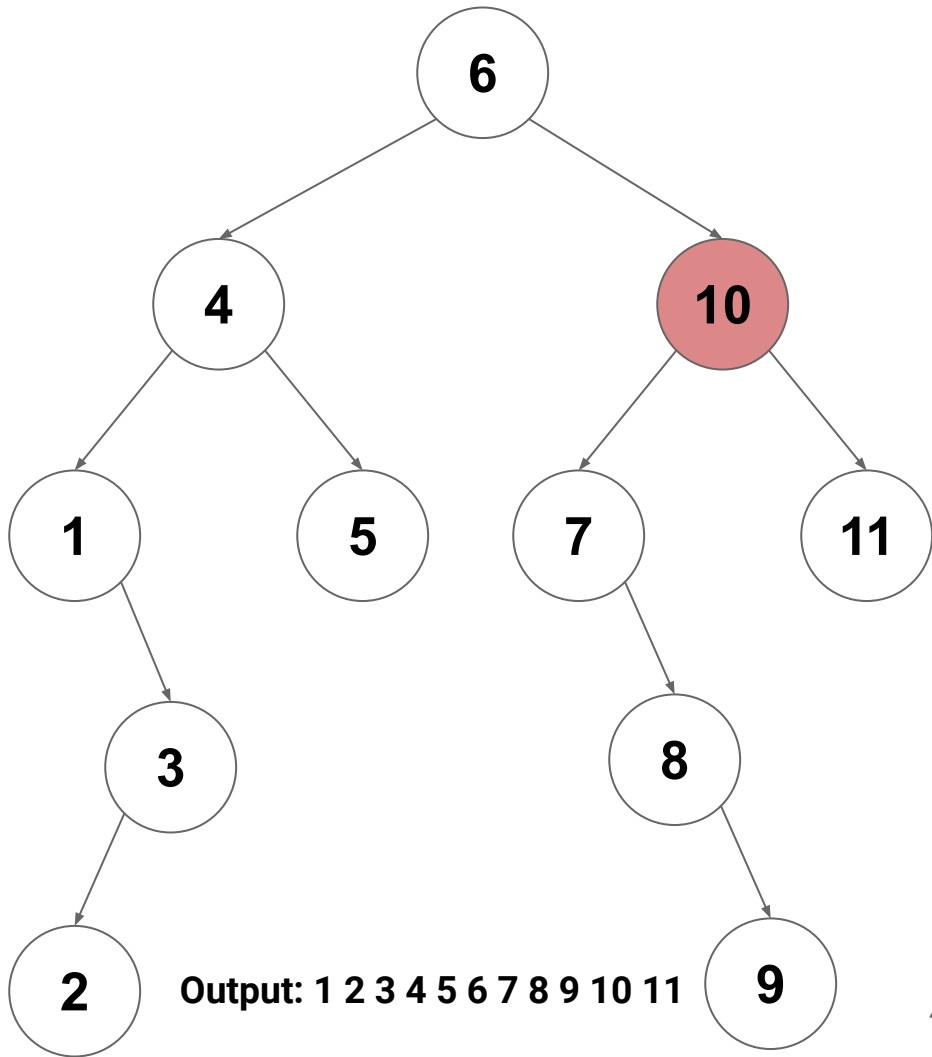
`visit(11)`



# In-Order Traversal on a BST

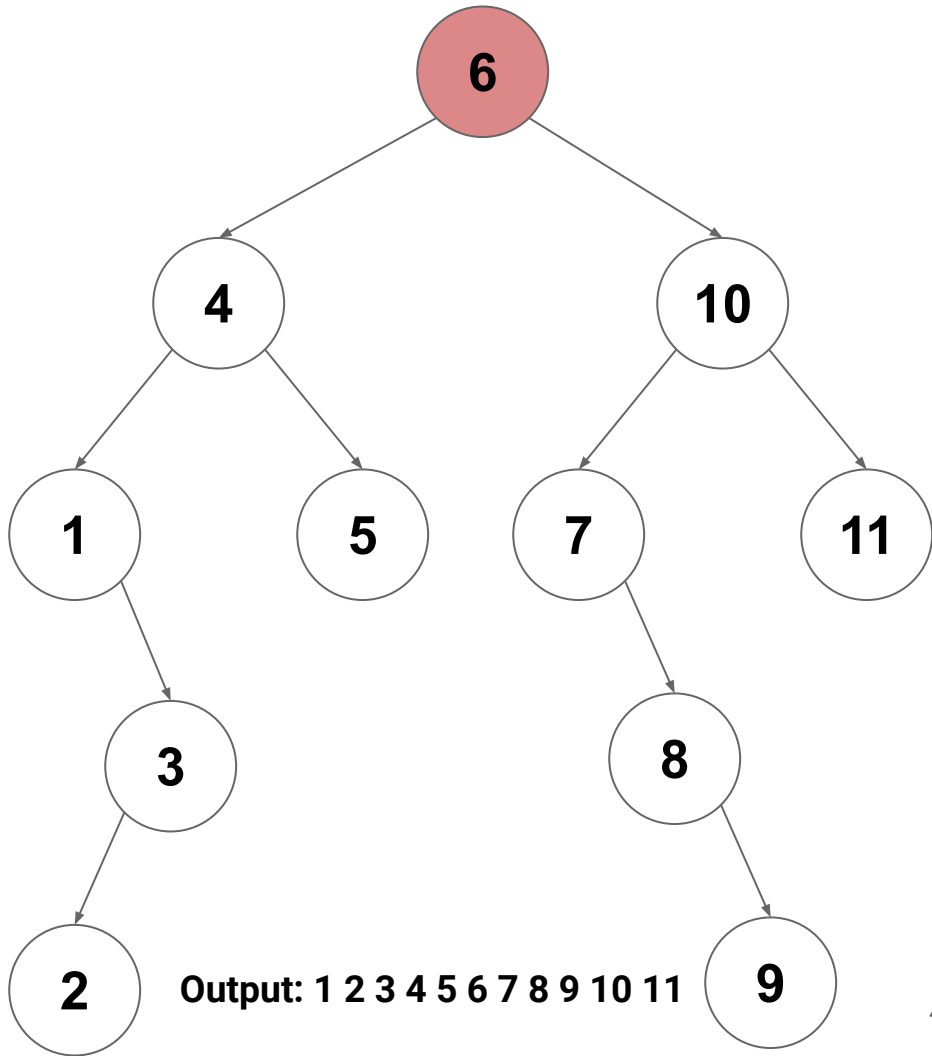
`inorderVisit(6)`

`inorderVisit(10)`

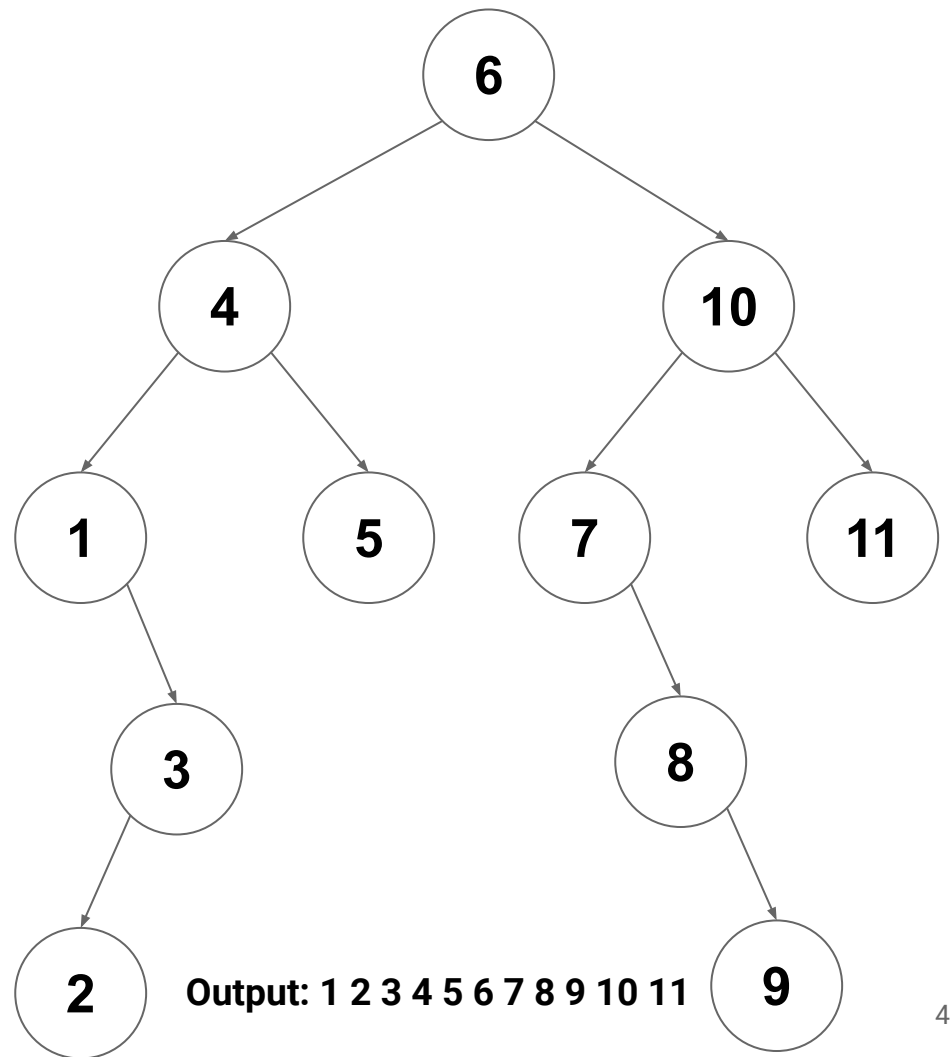


# In-Order Traversal on a BST

`inorderVisit(6)`



# In-Order Traversal on a BST



# Tree Traversal: In-Order Iterator

```
1 class TreeIterator<T> implements Iterator<T> {
2     Stack<TreeNode<T>> toVisit;
3     TreeIterator() {
4         toVisit = new Stack<>();
5         pushLeft(root);
6     }
7     void pushLeft(Optional<TreeNode<T>> node) {
8         if (node.isPresent()) {
9             toVisit.push(node.get());
10            pushLeft(node.get().leftChild());
11        }
12    }
13    /* ... */
14 }
```

# Tree Traversal: In-Order Iterator

```
1 class TreeIterator<T> implements Iterator<T> {
2     Stack<TreeNode<T>> toVisit;    Keep track of what we need to visit in a stack
3     TreeIterator() {
4         toVisit = new Stack<>();    Recursively push all the left nodes (they are
5         pushLeft(root);            the smallest and therefore should be visited
6     }                                first)
7     void pushLeft(Optional<TreeNode<T>> node) {
8         if (node.isPresent()) {
9             toVisit.push(node.get());
10            pushLeft(node.get().leftChild());
11        }
12    }
13    /* ... */
14 }
```

# Tree Traversal: In-Order Iterator

```
1 class TreeIterator<T> implements Iterator<T> {
2     Stack<TreeNode<T>> toVisit;
3     TreeIterator() {
4         toVisit = new Stack<>();
5         pushLeft(root);
6     }
7     void pushLeft(Optional<TreeNode<T>> node) {
8         if (node.isPresent()) {
9             toVisit.push(node.get());
10            pushLeft(node.get().leftChild());
11        }
12    }
13    /* ... */
14 }
```

Push the node, and then recursively push it's left children

# Tree Traversal: In-Order Iterator

```
1 class TreeIterator<T> implements Iterator<T> {
2     /* ... */
3
4     boolean hasNext() { return !toVisit.isEmpty(); }
5
6     T next() {
7         TreeNode<T> nextNode = toVisit.pop();
8         pushLeft(nextNode.rightChild);
9         return nextNode.value;
10    }
11 }
```

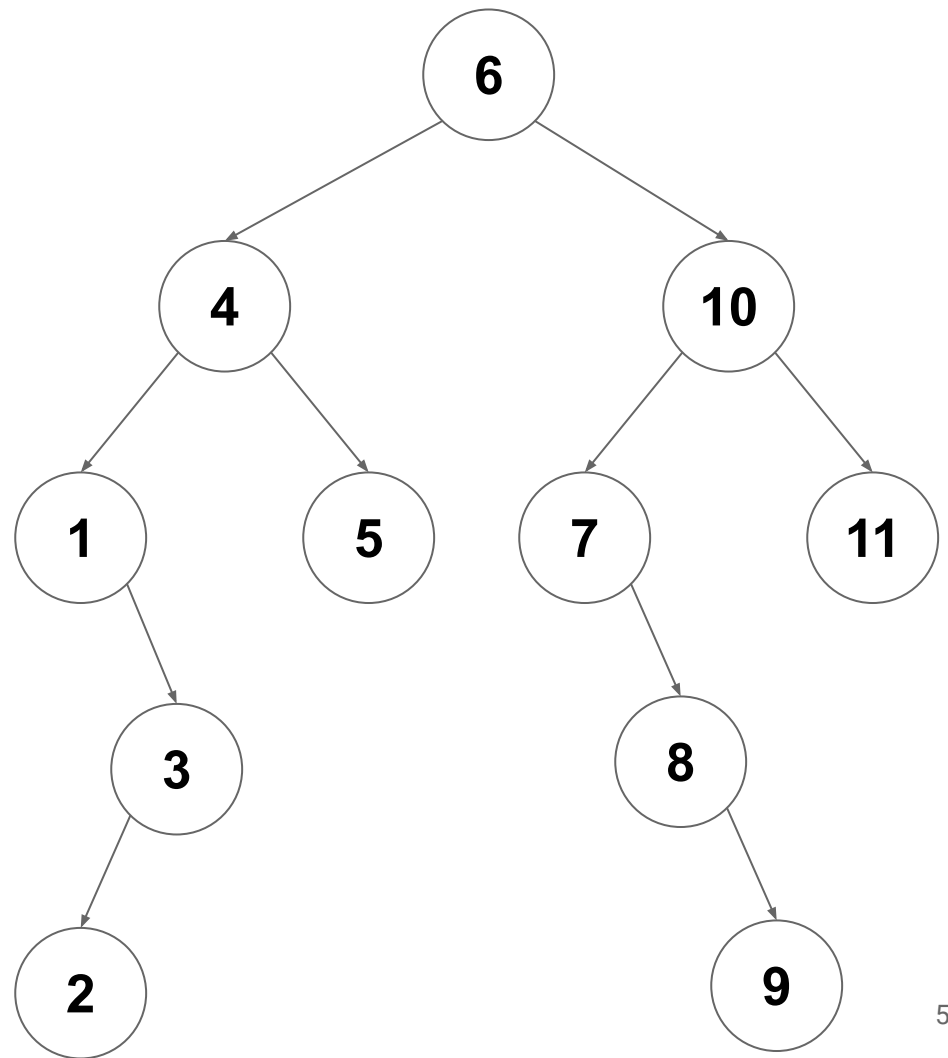


# Tree Traversal: In-Order Iterator

```
1 class TreeIterator<T> implements Iterator<T> {  
2     /* ... */  
3  
4     boolean hasNext() { return !toVisit.isEmpty(); }  
5  
6     T next() {  
7         TreeNode<T> nextNode = toVisit.pop();  
8         pushLeft(nextNode.rightChild);  
9         return nextNode.value;  
10    }  
11 }
```

Pop the next node, then push the left children of it's right subtree

# In-Order Traversal with an Iterator

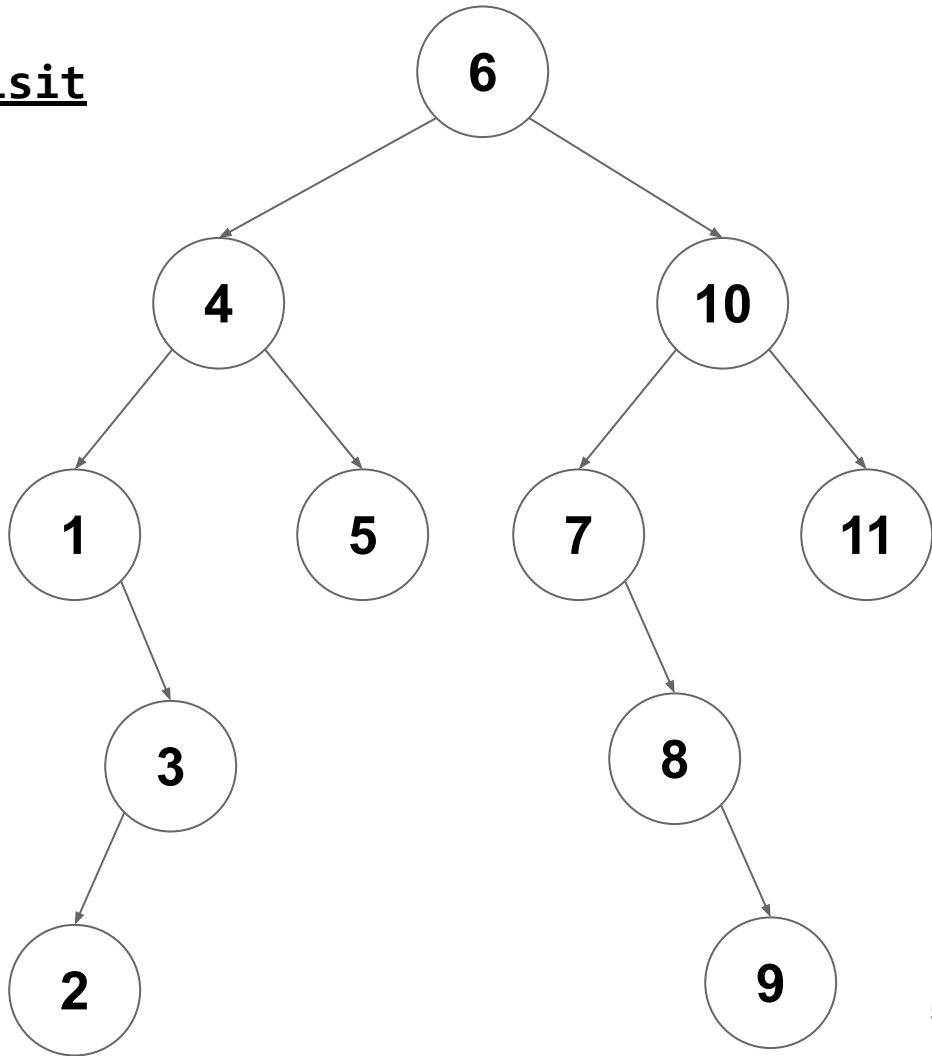


# In-Order Traversal with an Iterator

When we create the iterator, the `toVisit` stack is initialized

toVisit

6  
4  
1



# In-Order Traversal with an Iterator

next pops the stack (1),  
and calls pushLeft on  
the right subtree of 1

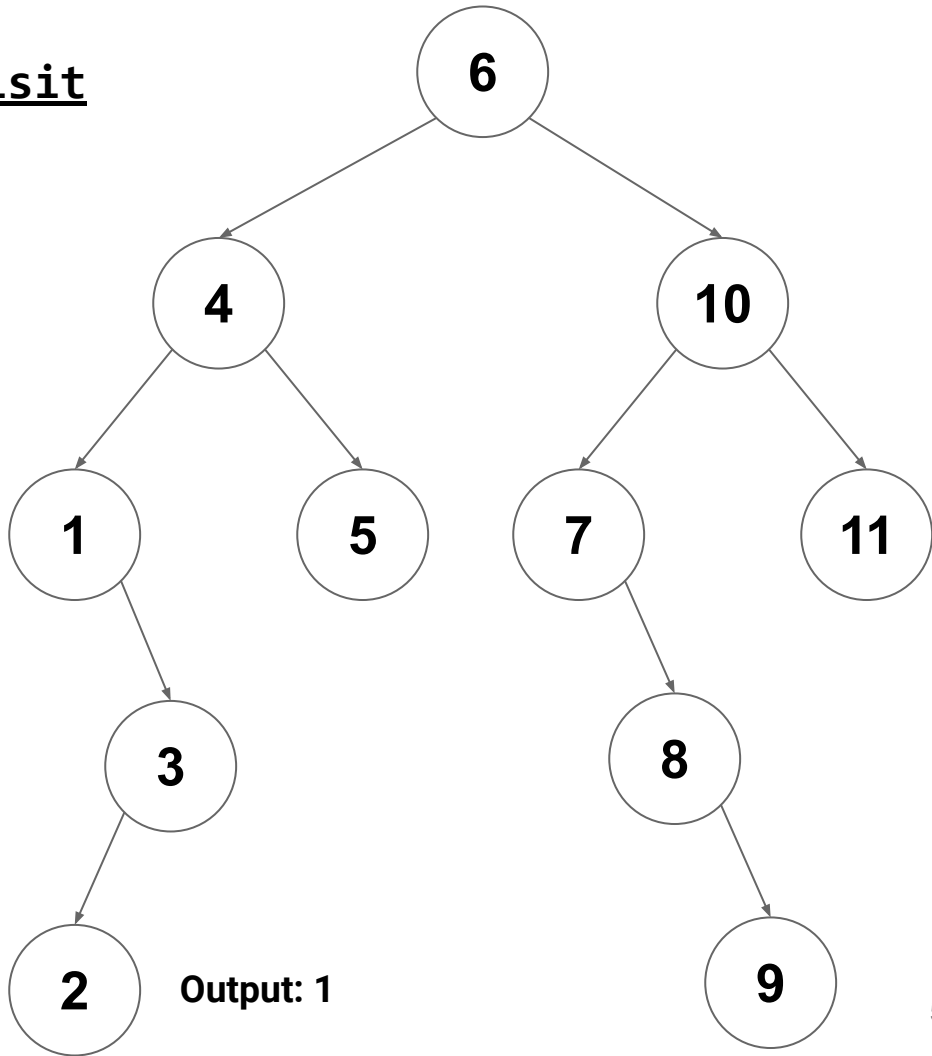
toVisit

6

4

3

2

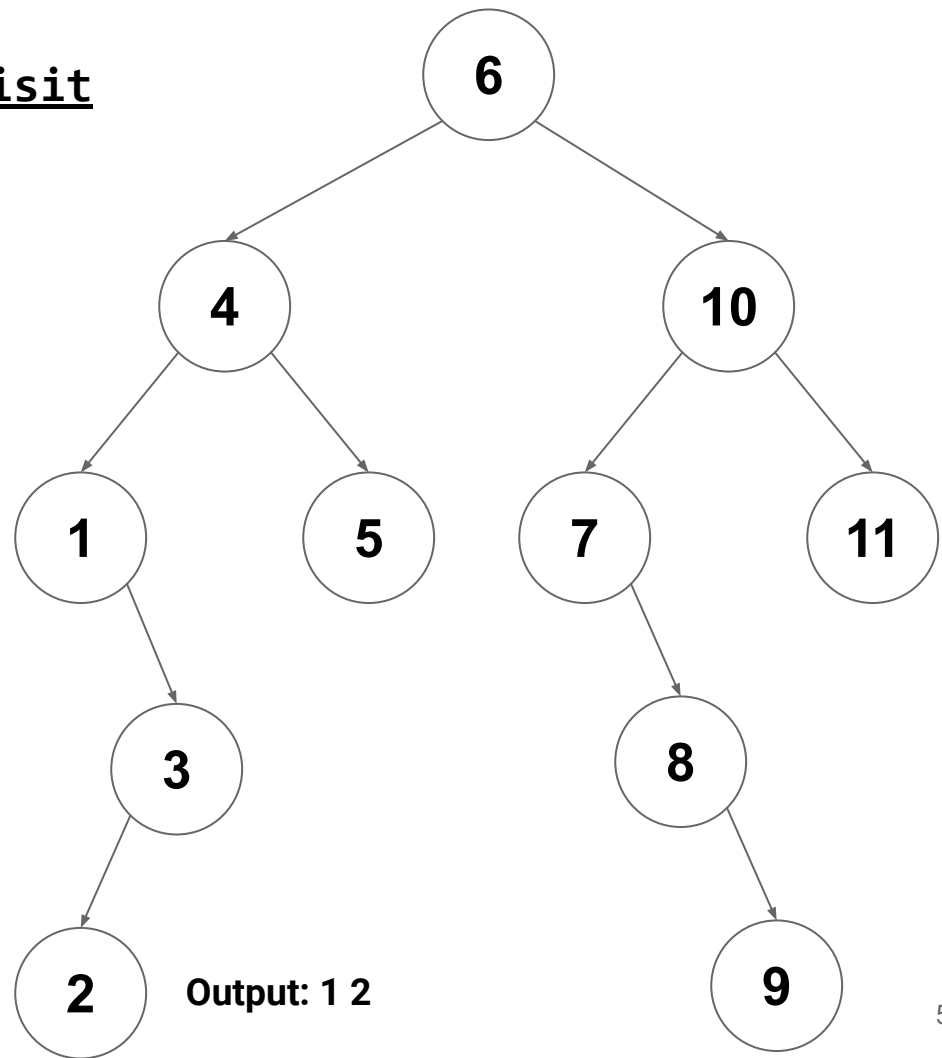


# In-Order Traversal with an Iterator

next pops the stack (2)  
and pushes the right  
subtree (nothing)

toVisit

6  
4  
3



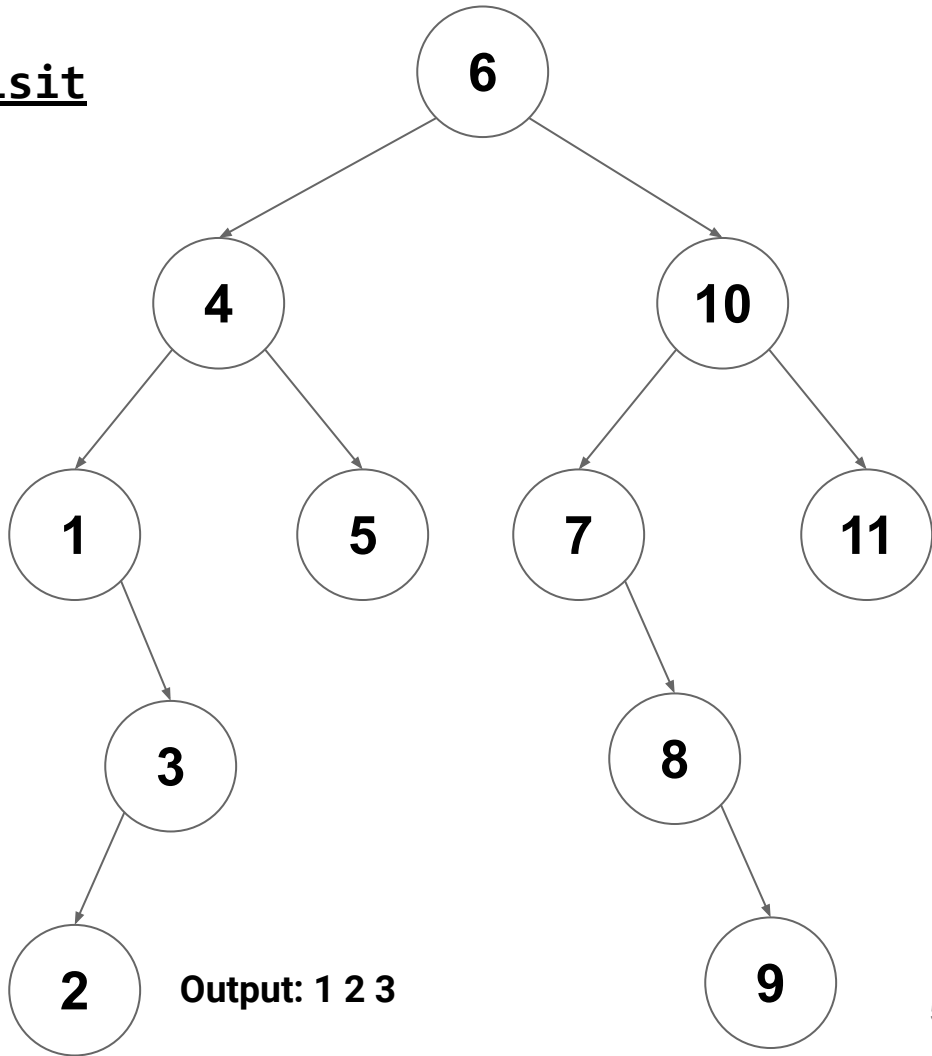
# In-Order Traversal with an Iterator

next pops the stack (3)  
and pushes the right  
subtree (nothing)

toVisit

6

4



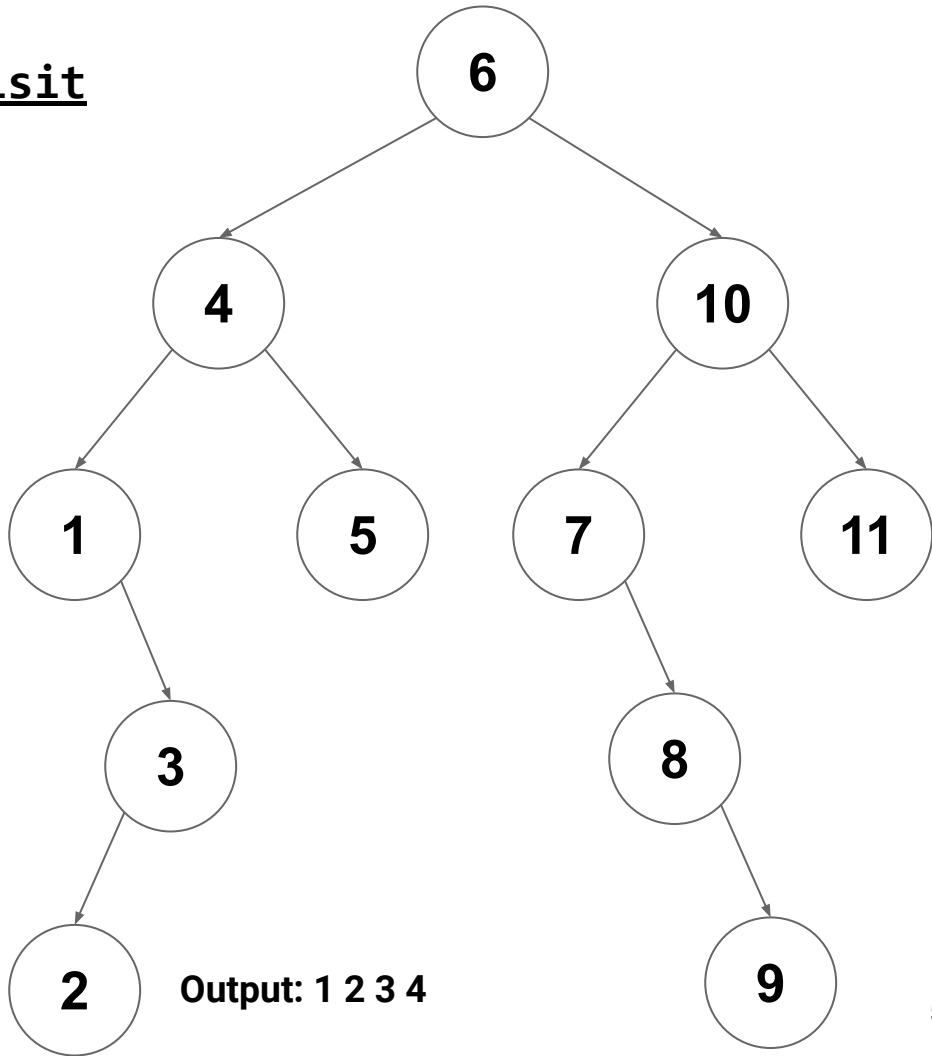
# In-Order Traversal with an Iterator

next pops the stack (4)  
and pushes the right  
subtree

toVisit

6

5

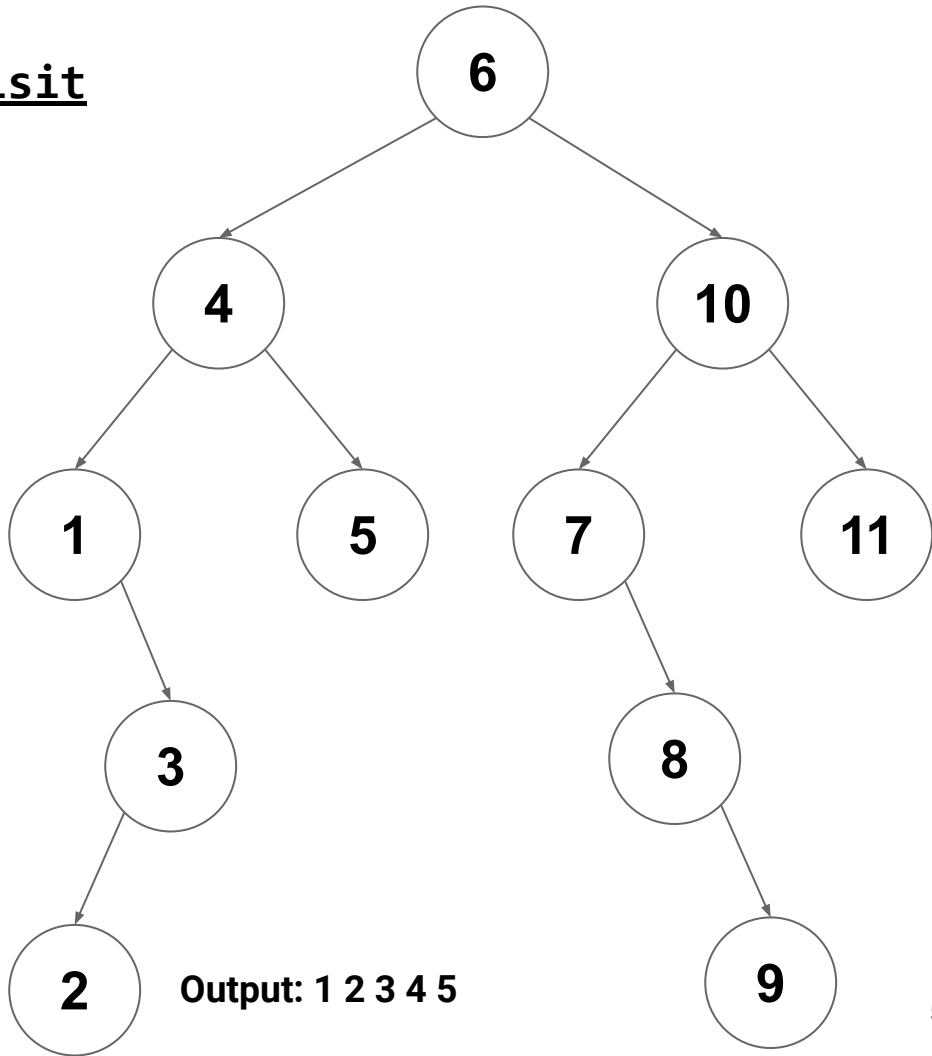


# In-Order Traversal with an Iterator

next pops the stack (5)  
and pushes the right  
subtree (nothing)

toVisit

6





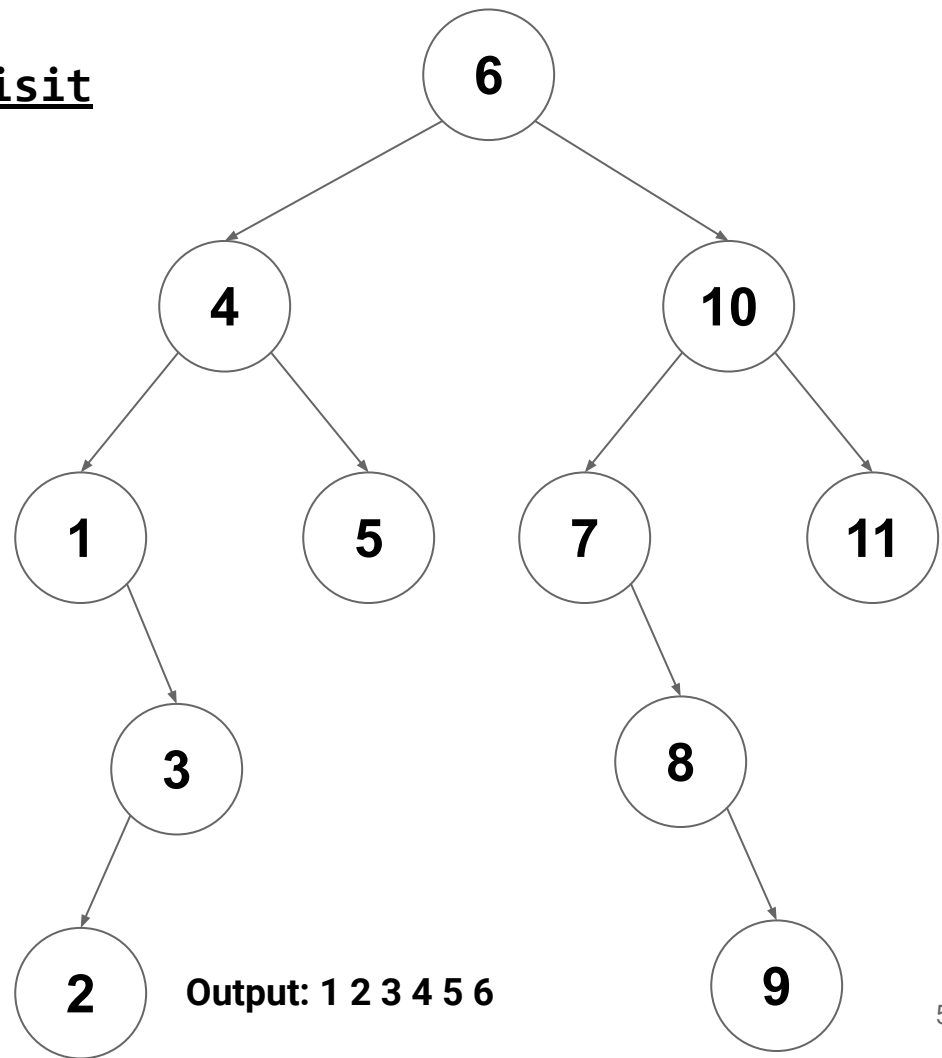
# In-Order Traversal with an Iterator

next pops the stack (6)  
and pushes the right  
subtree (10 7)

toVisit

10

7



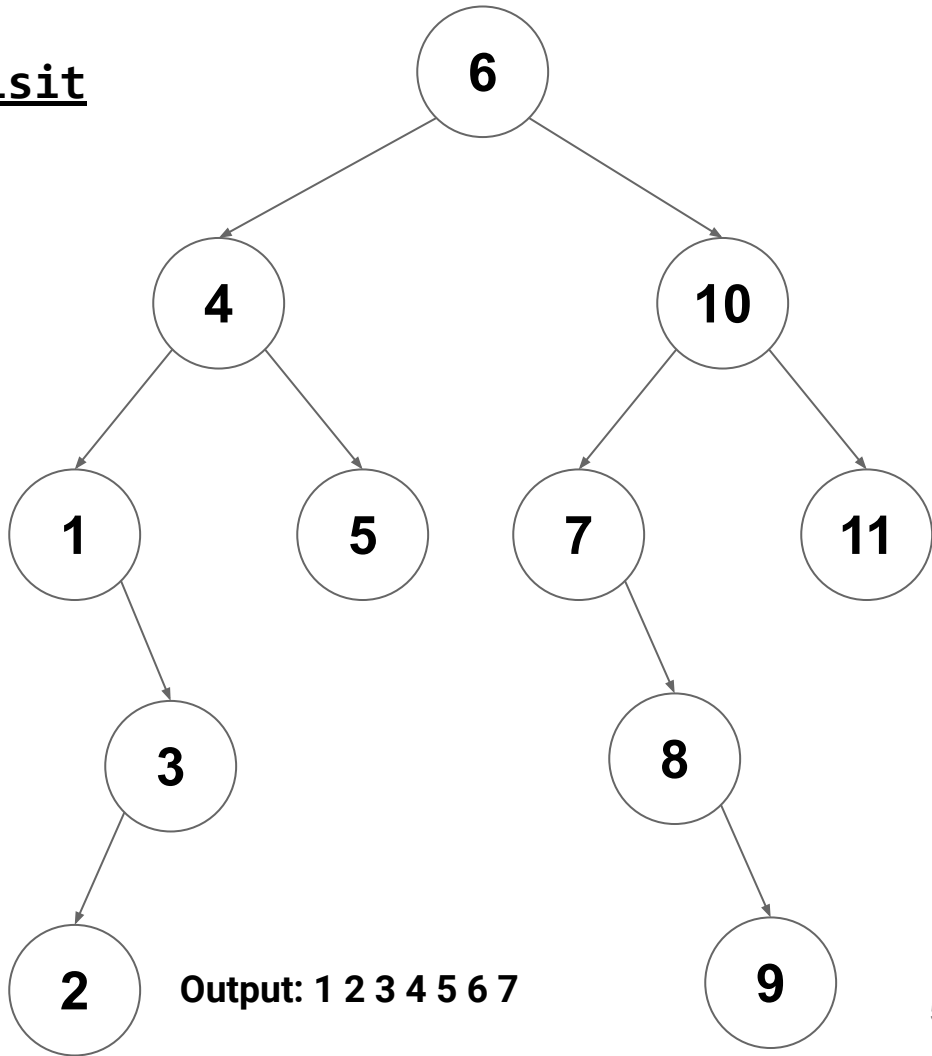
# In-Order Traversal with an Iterator

next pops the stack (7)  
and pushes the right  
subtree (8)

toVisit

10

8



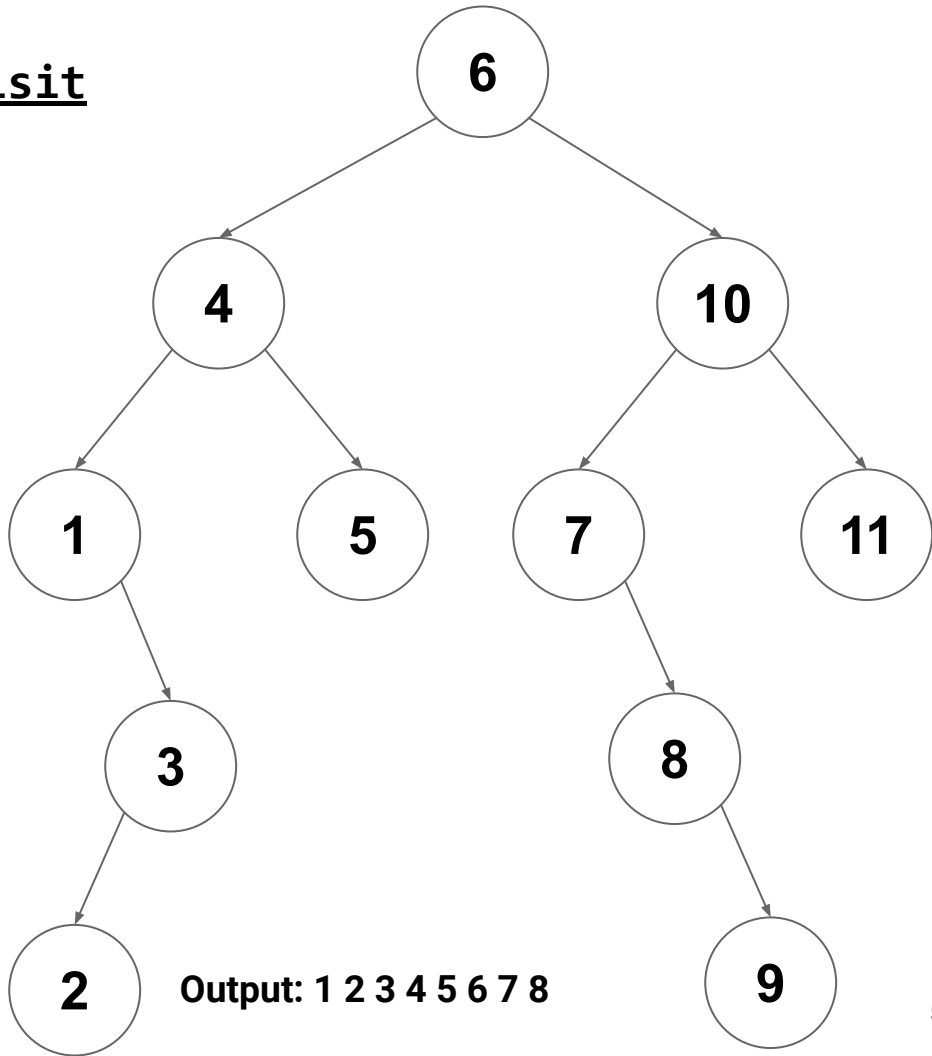
# In-Order Traversal with an Iterator

next pops the stack (8)  
and pushes the right  
subtree (9)

toVisit

10

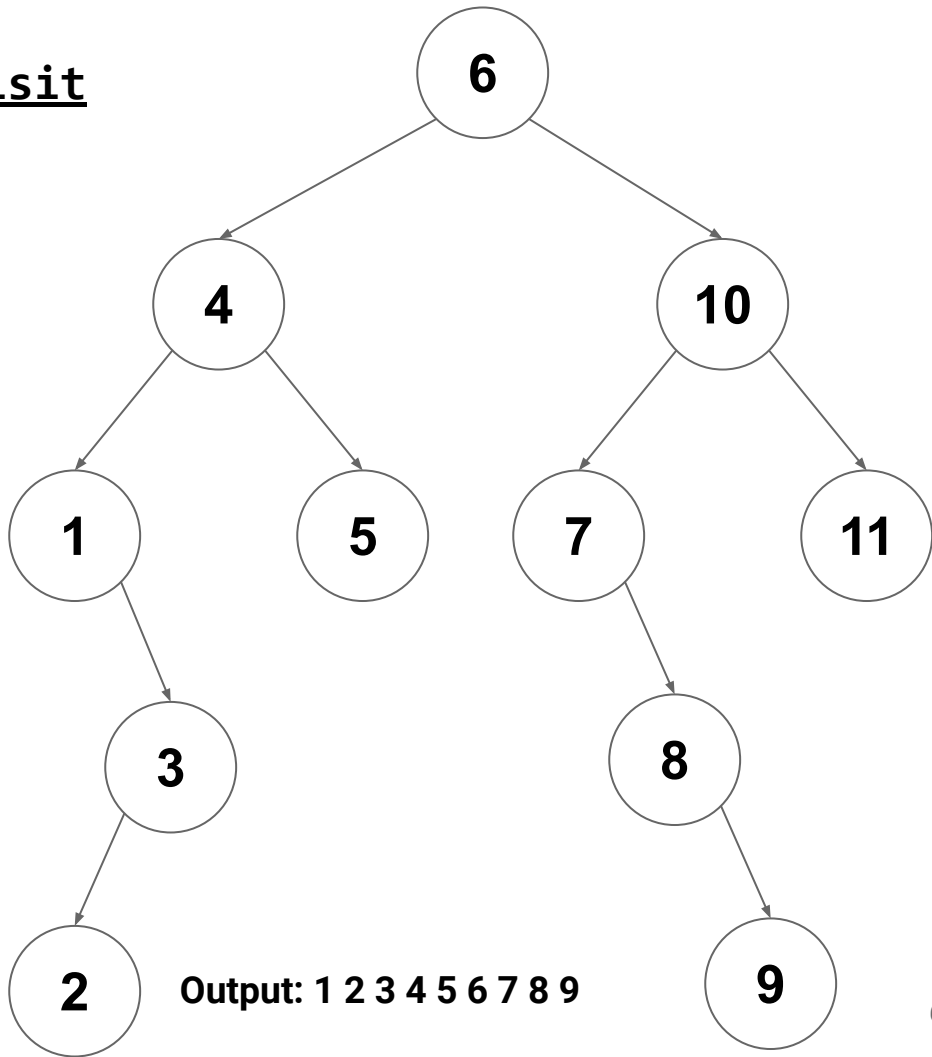
9



# In-Order Traversal with an Iterator

next pops the stack (9)  
and pushes the right  
subtree (nothing)

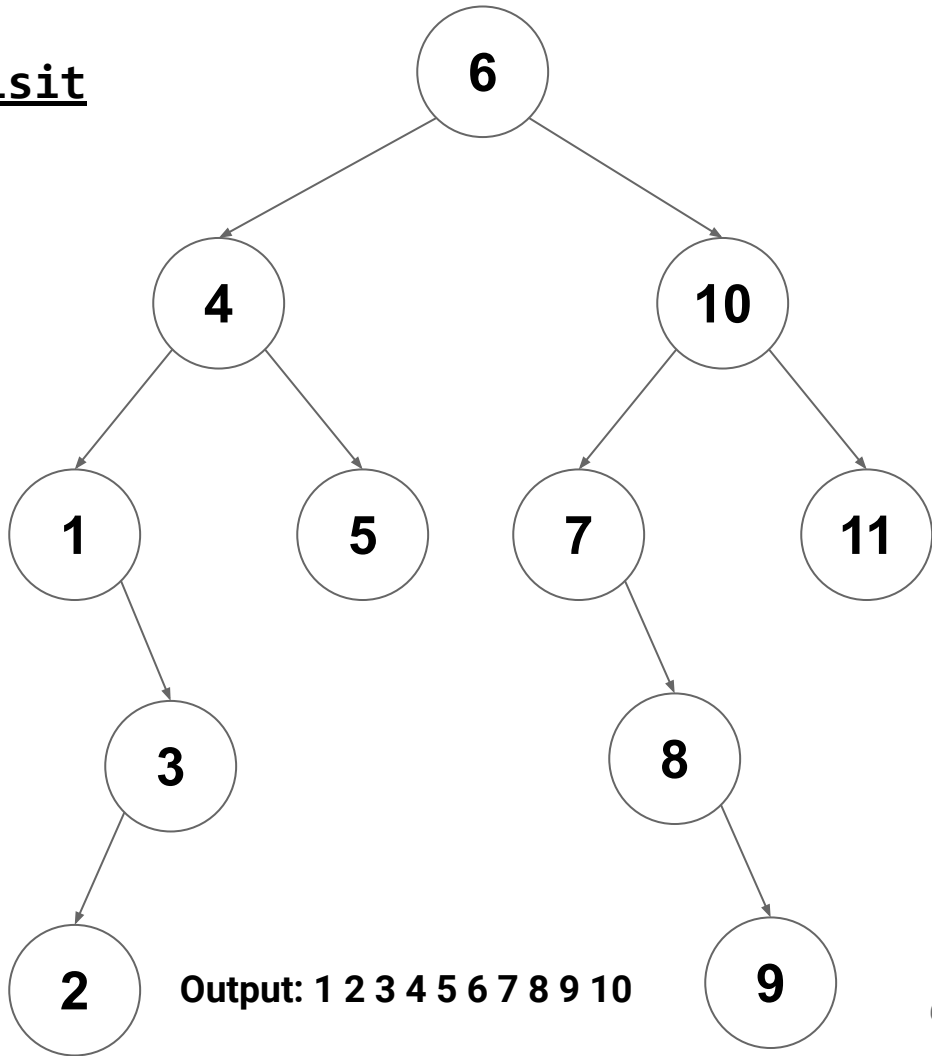
toVisit  
10



# In-Order Traversal with an Iterator

next pops the stack (10)  
and pushes the right  
subtree (11)

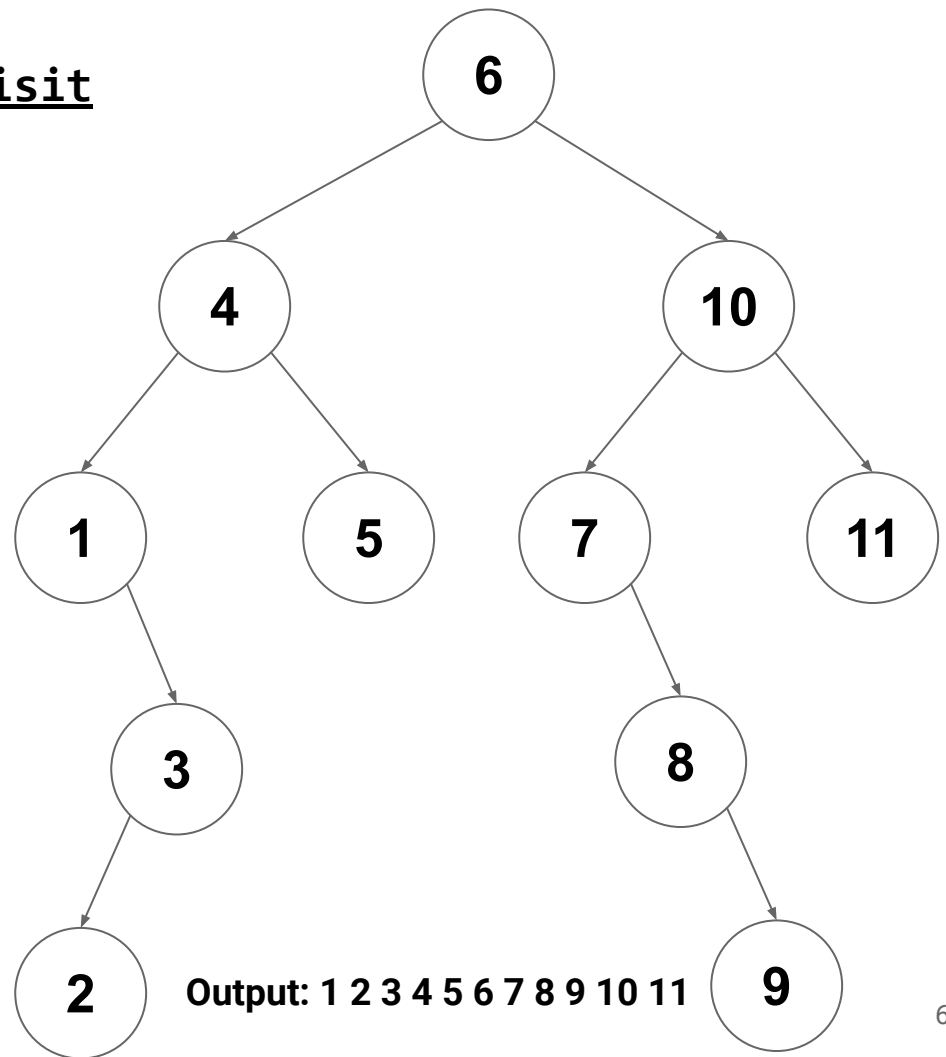
toVisit  
11



# In-Order Traversal with an Iterator

next pops the stack (11)  
and pushes the right  
subtree (nothing)

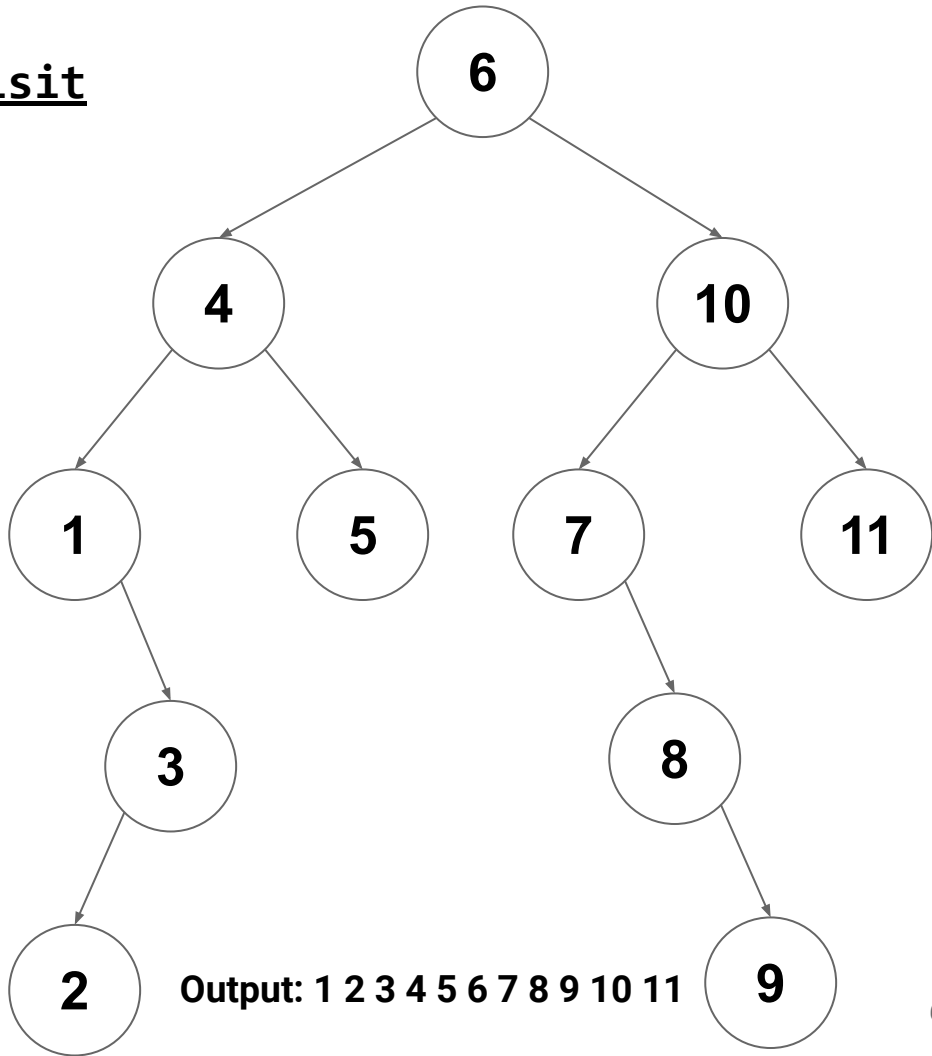
toVisit



# In-Order Traversal with an Iterator

Our `toVisit` stack is empty, so `isEmpty` will now be true

toVisit



# Complexity

```
1 TreeIterator() {  
2     toVisit = new Stack<>();  
3     pushLeft(root);  
4 }
```

*What is our worst-case runtime to initialize the iterator?*



# Complexity

```
1 TreeIterator() {  
2     toVisit = new Stack<>();  
3     pushLeft(root);  
4 }
```

*What is our worst-case runtime to initialize the iterator?  $O(d)$*

# Complexity

```
1 TreeIterator() {  
2     toVisit = new Stack<>();  
3     pushLeft(root);  
4 }
```

*What is our worst-case runtime to initialize the iterator?  $O(d)$*

*(we may have to push as many as  $d$  nodes onto the stack)*

# Complexity

```
1 T next() {  
2   TreeNode<T> nextNode = toVisit.pop();  
3   pushLeft(nextNode.rightChild);  
4   return nextNode.value;  
5 }
```

*What is our worst-case runtime to call next?*

# Complexity

```
1 T next() {  
2   TreeNode<T> nextNode = toVisit.pop();  
3   pushLeft(nextNode.rightChild);  
4   return nextNode.value;  
5 }
```

*What is our worst-case runtime to call next?  $O(d)$*

*(we may have to push as many as  $d$  nodes onto the stack)*

# Complexity

*What is the worst-case complexity to visit ALL  $n$  nodes?*

# Complexity

*What is the worst-case complexity to visit ALL  $n$  nodes?*

**Each node is at the top of the stack exactly once:**

# Complexity

*What is the worst-case complexity to visit ALL  $n$  nodes?*

**Each node is at the top of the stack exactly once:**

- One push  **$O(1)$**

# Complexity

*What is the worst-case complexity to visit ALL  $n$  nodes?*

**Each node is at the top of the stack exactly once:**

- One push  $O(1)$
- One pop  $O(1)$



# Complexity

*What is the worst-case complexity to visit ALL  $n$  nodes?*

**Each node is at the top of the stack exactly once:**

- One push  $O(1)$
- One pop  $O(1)$

**Total:  $O(n)$**

# Balancing Trees

# BST Operations

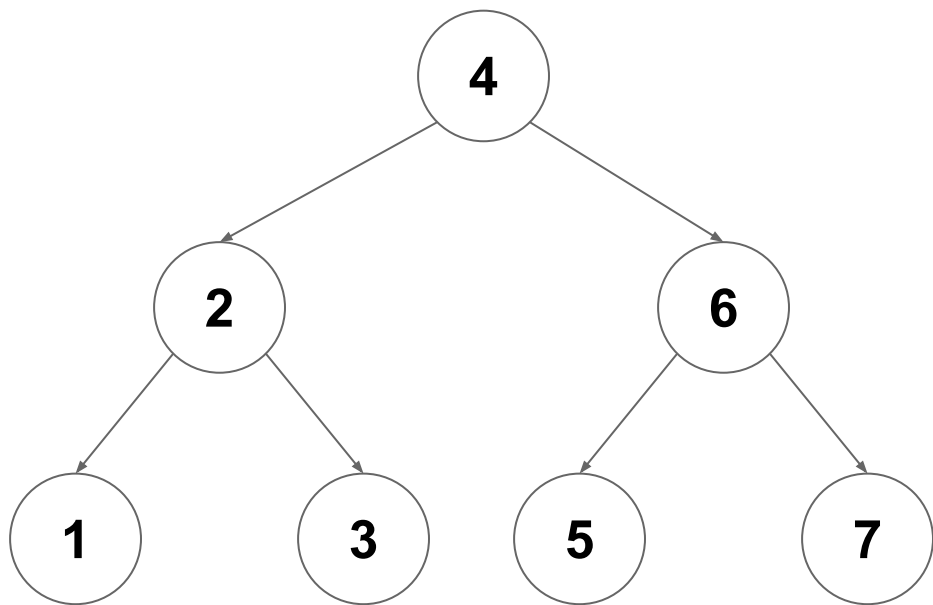
Operation	Runtime
<code>find</code>	$O(d)$
<code>insert</code>	$O(d)$
<code>remove</code>	$O(d)$

What is the runtime in terms of  $n$ ?  $O(n)$

$$\log(n) \leq d \leq n$$

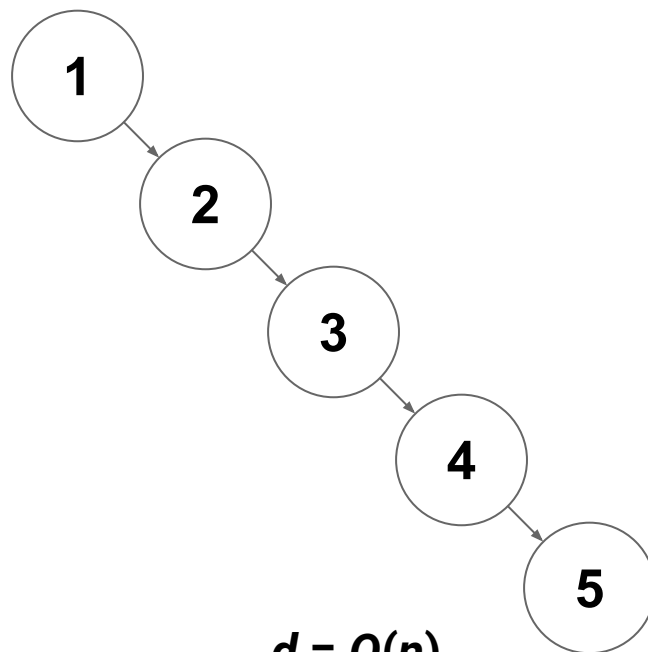
# Tree Depth vs Size

If  $\text{height}(\text{left}) \approx \text{height}(\text{right})$



$$d = O(\log(n))$$

If  $\text{height}(\text{left}) \ll \text{height}(\text{right})$



$$d = O(n)$$

# Balanced Trees

**Balanced Trees are good: Faster find, insert, remove**

# Balanced Trees

**Balanced Trees are good: Faster find, insert, remove**

*What do we mean by balanced?*

# Balanced Trees

**Balanced Trees are good: Faster find, insert, remove**

*What do we mean by balanced?*  **$|\text{height}(\text{right}) - \text{height}(\text{left})| \leq 1$**

# Balanced Trees

**Balanced Trees are good:** Faster `find`, `insert`, `remove`

*What do we mean by balanced?*  $|\text{height}(\text{right}) - \text{height}(\text{left})| \leq 1$

*How do we keep a tree balanced?*



# Balanced Trees - Two Approaches

## Option 1

Keep left/right subtrees within **+/-1** of each other in height

(add a field to track amount of "imbalance")

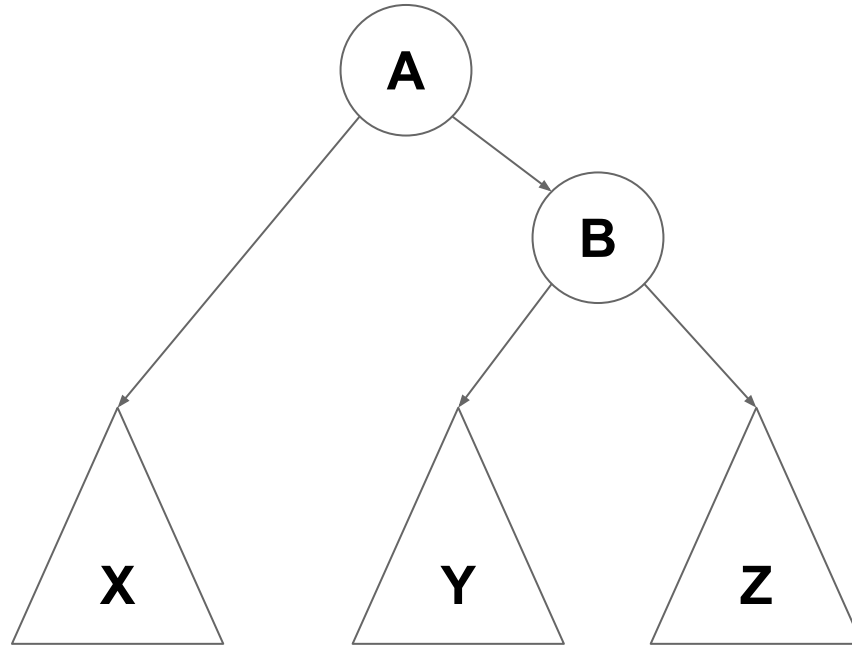
## Option 2

Keep leaves at some minimum depth ( **$d/2$** )

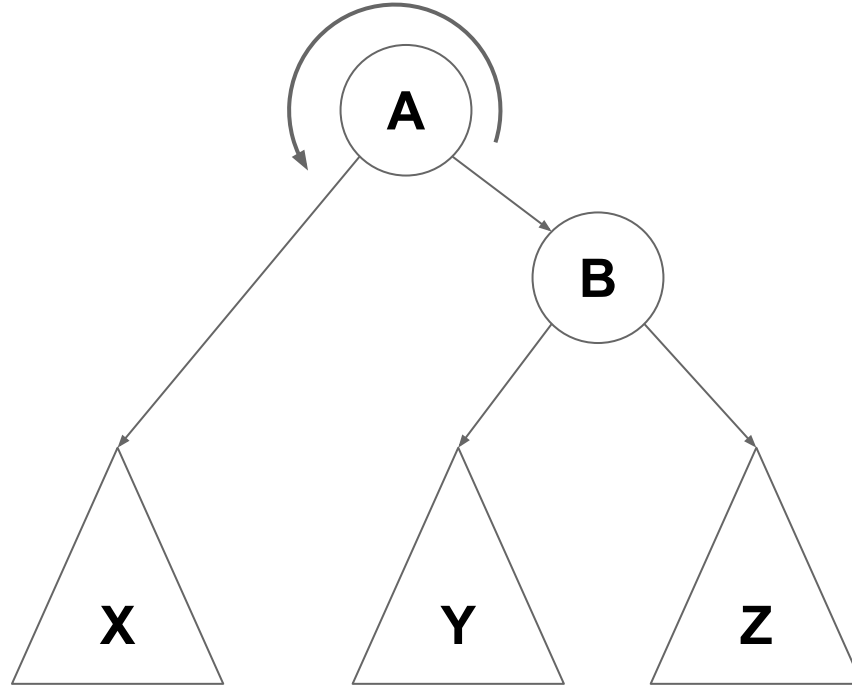
(Add a color to each node marking it as "red" or "black")

**Ok...but how do we enforce  
this...?**

# Rebalancing Trees (rotations)

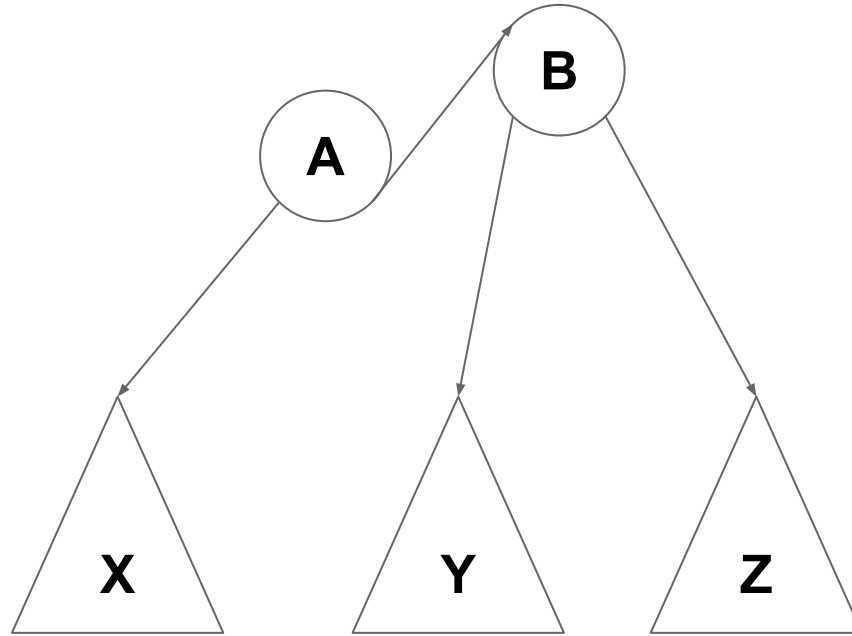


# Rebalancing Trees (rotations)



**Rotate(A, B)**

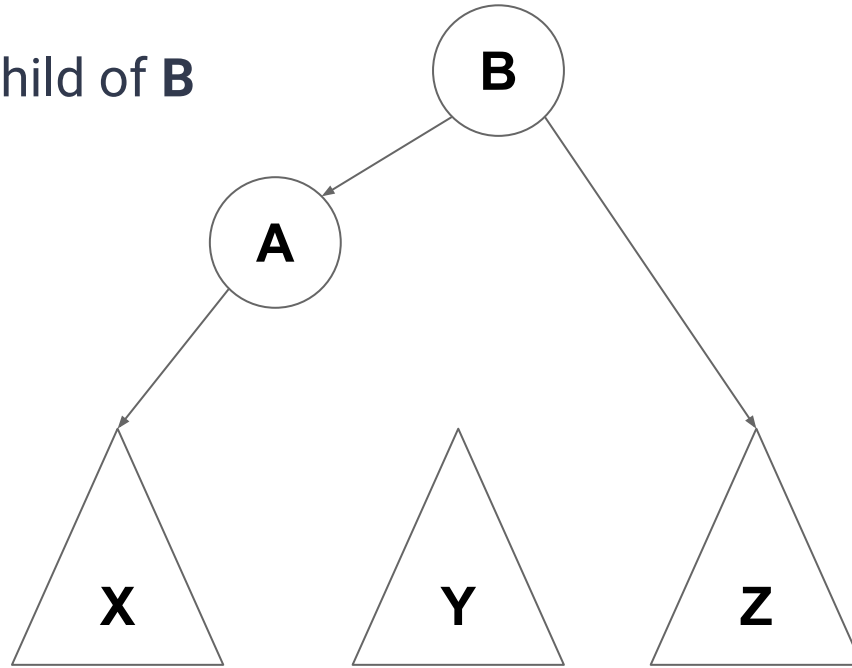
# Rebalancing Trees (rotations)



**Rotate(A, B)**

# Rebalancing Trees (rotations)

Make **A** the left child of **B**

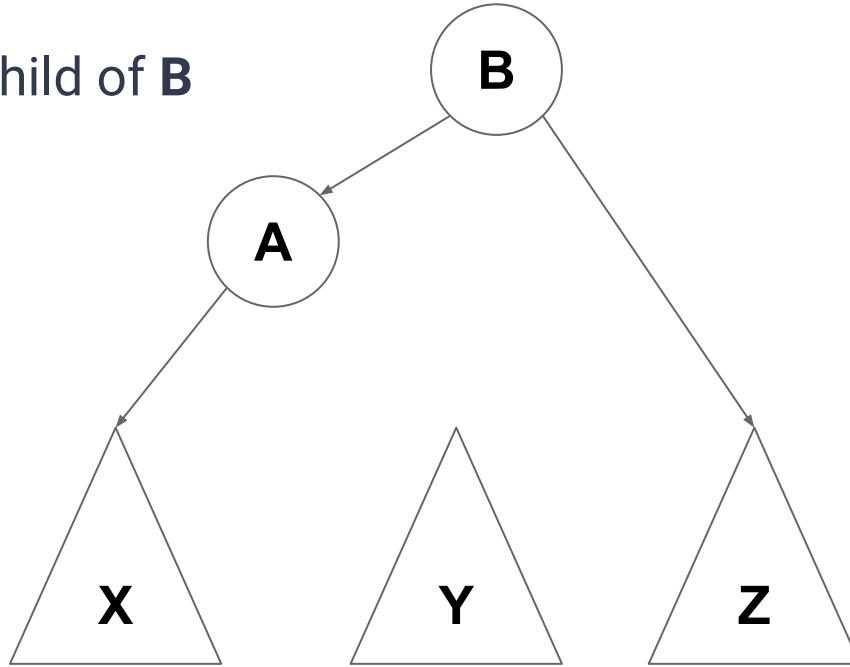


**Rotate(A, B)**

# Rebalancing Trees (rotations)

Make **A** the left child of **B**

What about **Y**?



**Rotate(A, B)**

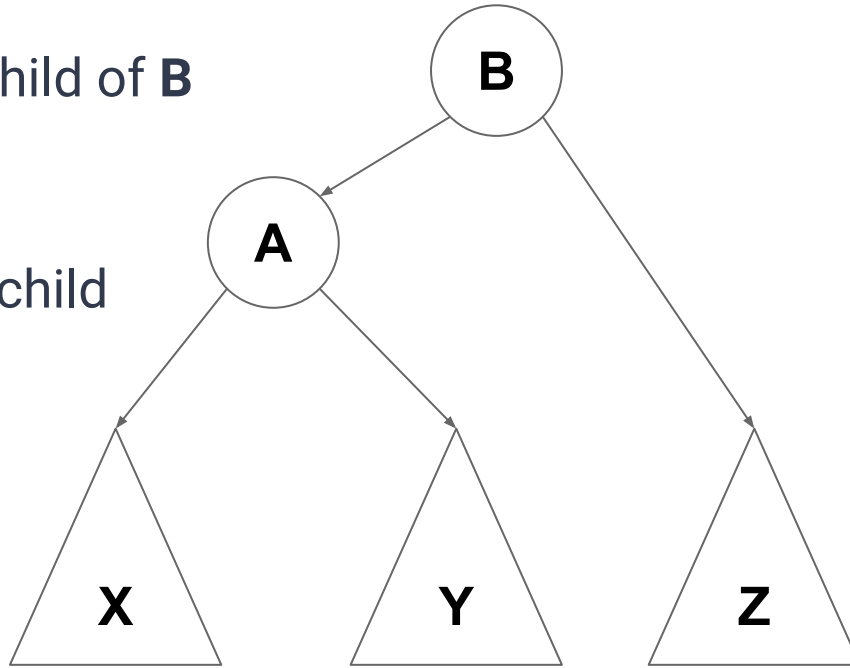
# Rebalancing Trees (rotations)

Make **A** the left child of **B**

What about **Y**?

Make it the right child

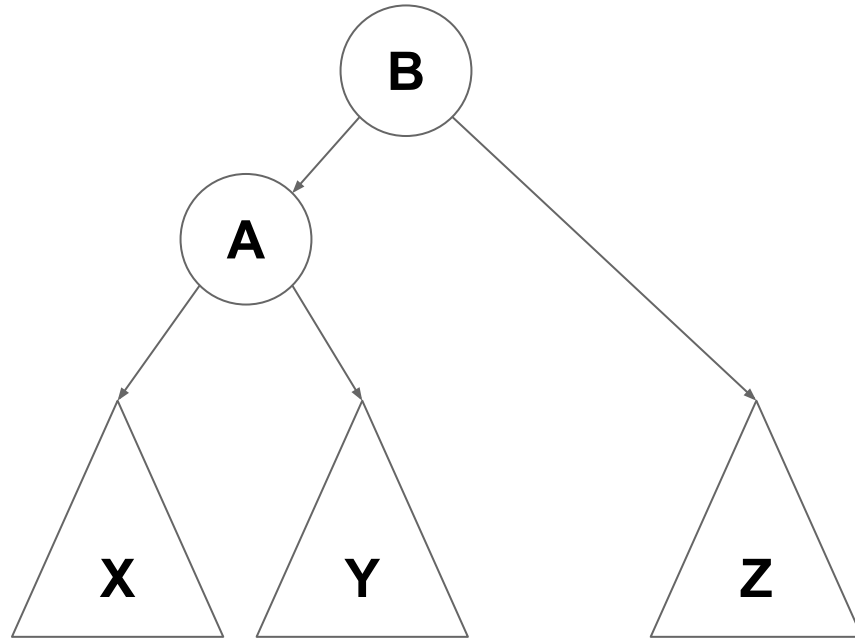
of **A**



**Rotate(A, B)**



# Rebalancing Trees (rotations)

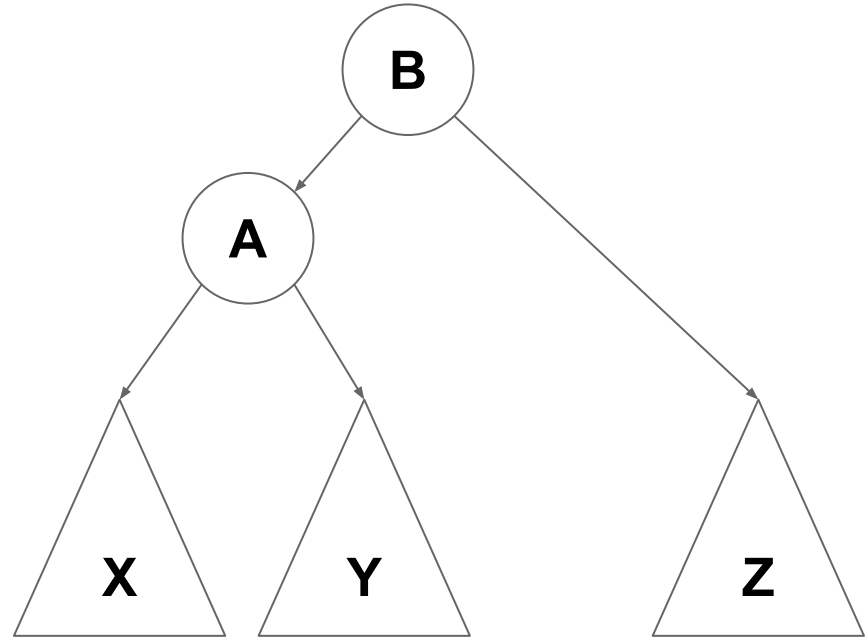


**Rotate(A, B)**

# Rebalancing Trees (rotations)

**A** became **B**'s left child

**B**'s left child became **A**'s right child



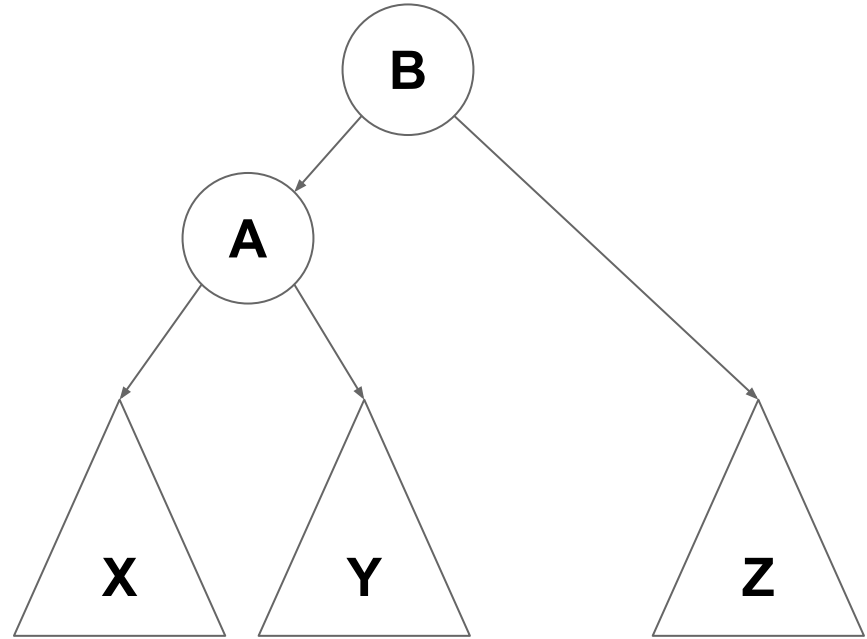
**Rotate(A, B)**

# Rebalancing Trees (rotations)

**A** became **B**'s left child

**B**'s left child became **A**'s right child

*Is ordering maintained?*



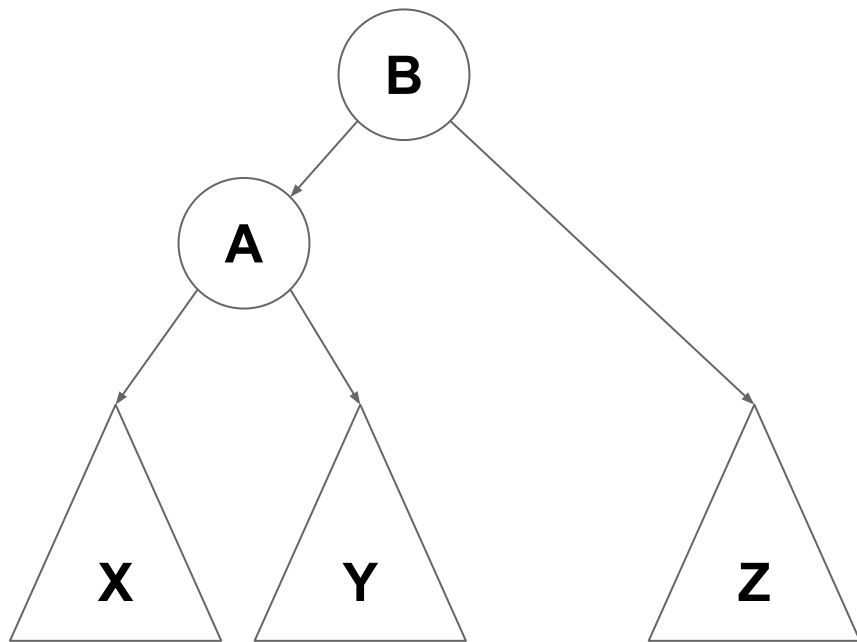
**Rotate(A, B)**

# Rebalancing Trees (rotations)

**A** became **B**'s left child

**B**'s left child became **A**'s right child

*Is ordering maintained? Yes!*



**Rotate(A, B)**

# Rebalancing Trees (rotations)

**A** became **B**'s left child

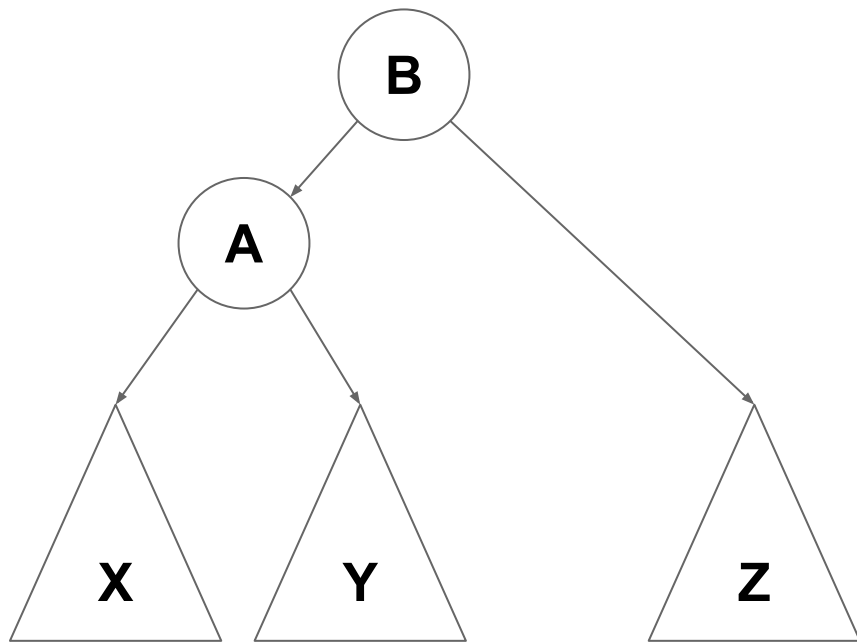
**B**'s left child became **A**'s right child

*Is ordering maintained? Yes!*

**B** used to be the right child of **A**

Therefore **B** is bigger than **A**

Therefore **A** is smaller than **B** ✓



**Rotate(A, B)**

# Rebalancing Trees (rotations)

**A** became **B**'s left child

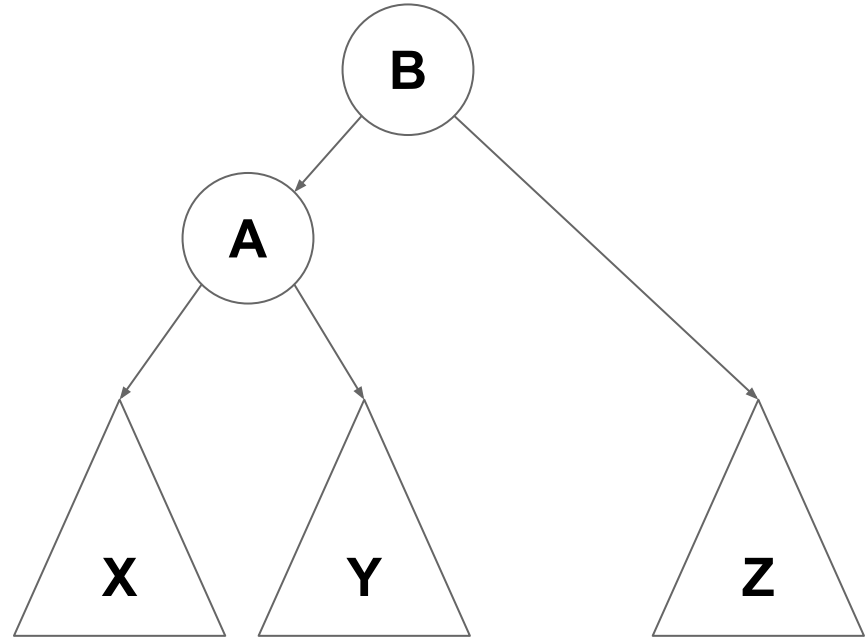
**B**'s left child became **A**'s right child

*Is ordering maintained? Yes!*

**Y** used to be in the left subtree of **B**

Therefore **Y** is smaller than **B**

It is still left of **B** ✓



**Rotate(A, B)**

# Rebalancing Trees (rotations)

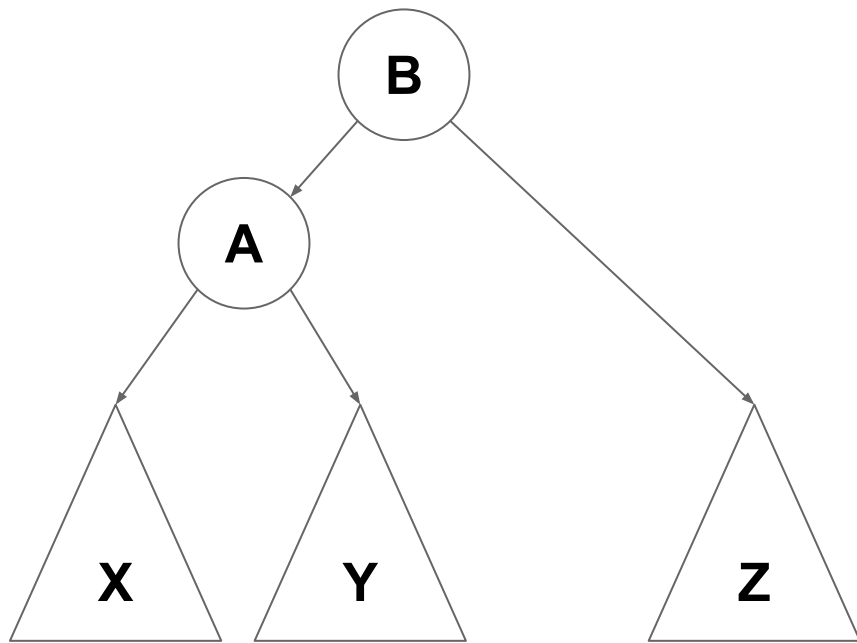
**A** became **B**'s left child

**B**'s left child became **A**'s right child

*Is ordering maintained? Yes!*

**Y** used to be in the right subtree of **A**

It is still in the right subtree of **A** ✓



**Rotate(A, B)**

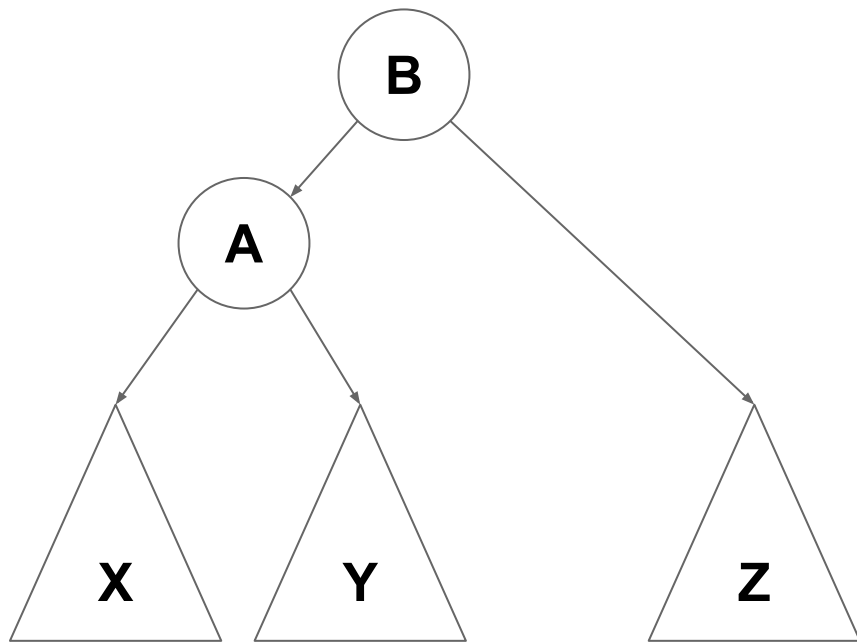
# Rebalancing Trees (rotations)

**A** became **B**'s left child

**B**'s left child became **A**'s right child

*Is ordering maintained? Yes!*

*Complexity?*



**Rotate(A, B)**



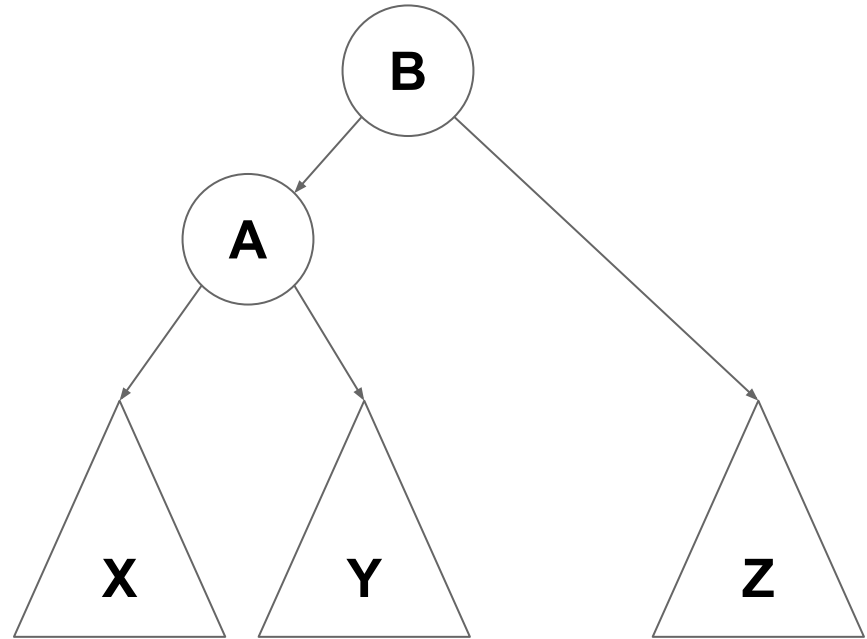
# Rebalancing Trees (rotations)

**A** became **B**'s left child

**B**'s left child became **A**'s right child

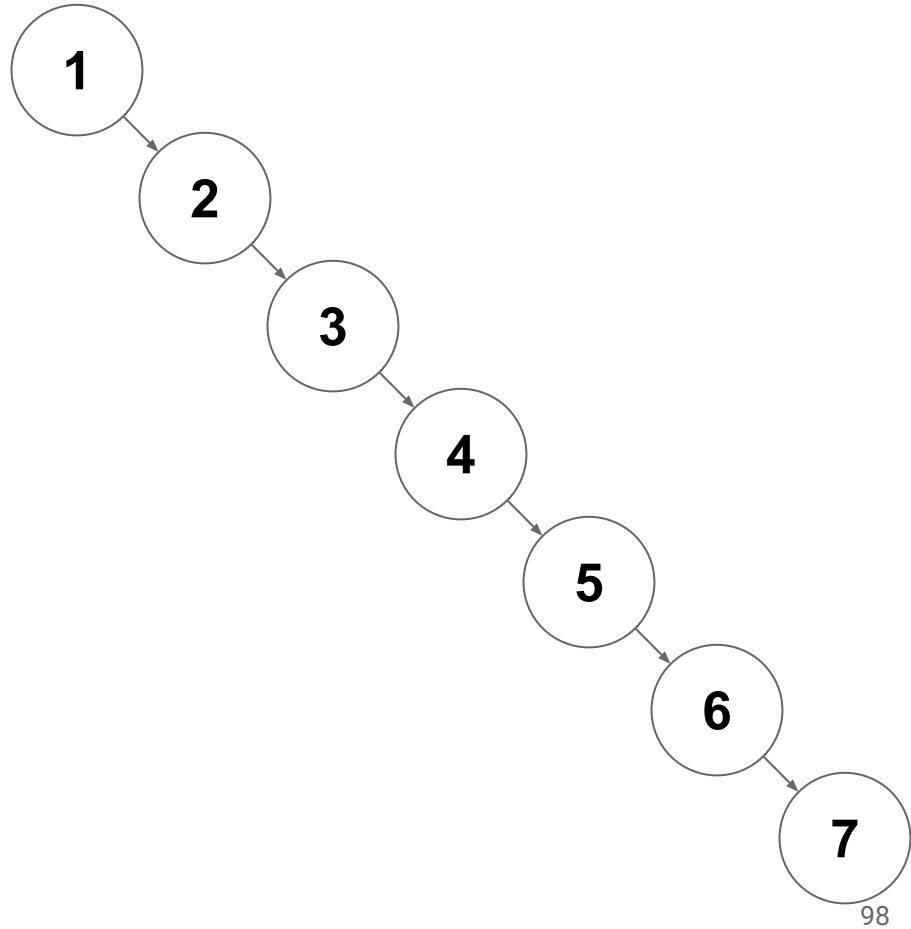
*Is ordering maintained? Yes!*

*Complexity?  $O(1)$*



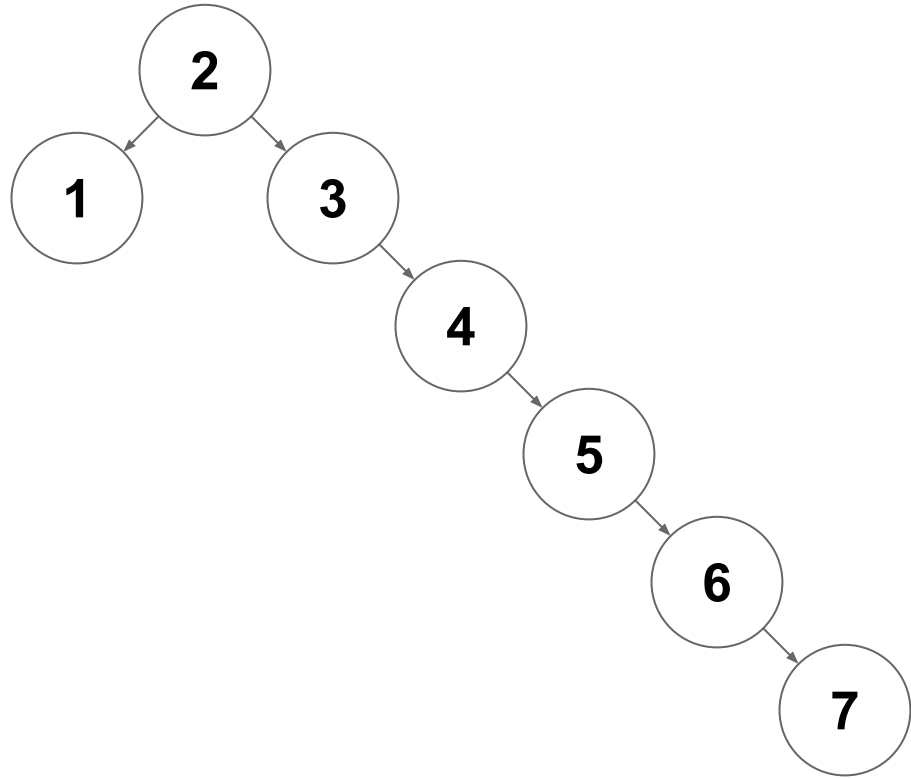
**Rotate(A, B)**

# Rebalancing Trees



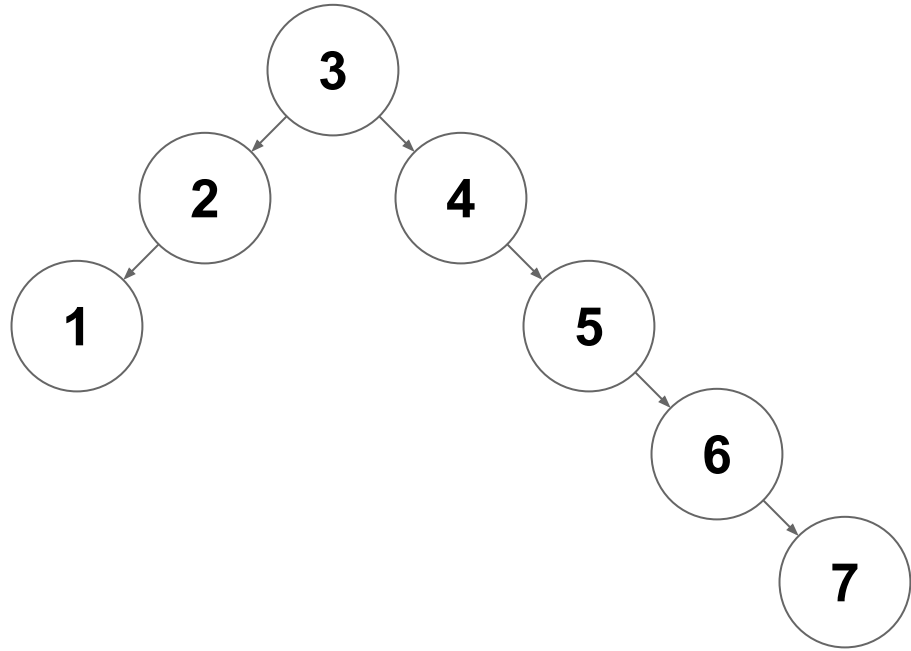
# Rebalancing Trees

Rotate(1,2)



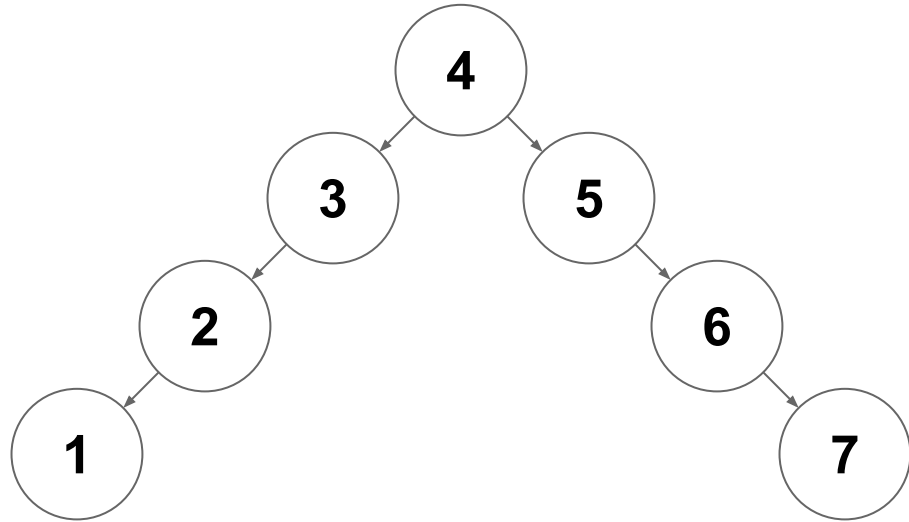
# Rebalancing Trees

Rotate(2,3)



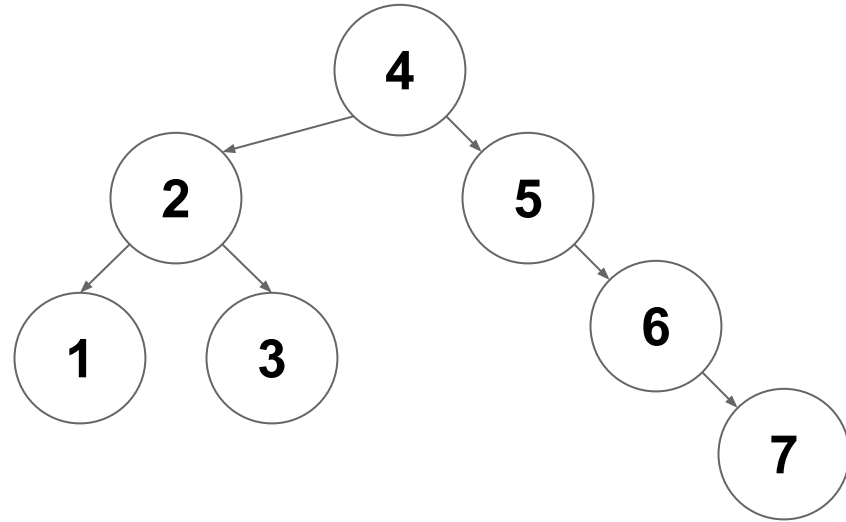
# Rebalancing Trees

Rotate(3,4)



# Rebalancing Trees

Rotate(3,2)



# Rebalancing Trees

Rotate(5,6)

